"C-Level" Programming of Parallel Coprocessor Accelerators

Benjamin Ylvisaker

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2010

Program Authorized to Offer Degree: Computer Science and Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Benjamin Ylvisaker

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Co-Chairs of the Supervisory Committee:

_____

William H.c. Ebeling

_____

Scott Hauck

Reading Committee:

_____

William H.c. Ebeling

_____

Scott Hauck

_____

Daniel Grossman

Date: _____

University of Washington

Abstract

"C-Level" Programming of Parallel Coprocessor Accelerators

Benjamin Ylvisaker

Co-Chairs of the Supervisory Committee:
Professor William H.c. Ebeling
Computer Science and Engineering

Professor Scott Hauck
Electrical Engineering

We believe that FPGA-like parallel coprocessor accelerators can be programmed efficiently at the "C level" of abstraction. In order to support this claim we define an abstract architectural model of accelerators that conveys the kind of high-level behavior and performance characteristics that the von Neumann model conveys to programmers of conventional processors. Using the model as a guide we define a programming language and compilation strategy that:

1. do not impose programming style restrictions that are not inherent in the model,
2. do not introduce serious inefficiencies, and
3. are performance portable across implementations of the model.

In this dissertation I describe C-level programming of accelerators broadly, and make three particular contributions to the programmability of accelerators.

- Enhanced loop flattening is a new method for translating loop nests with arbitrary static control flow into a form that can be efficiently pipelined with conventional algorithms designed for simple loops. This method advances the goal of supporting a wide set of programming styles with reasonable efficiency.

- Parallel accelerators have statically managed resources–like local memories–that vary widely in capacity from one implementation to the next. In order to get close to peak performance, applications must be tuned to the specific resources available in a given implementation, and empirical auto-tuning is an attractive way to do that. I propose and evaluate a new probabilistic auto-tuning method that elegantly handles situation where many possible configurations of the application fail to work at all because they exceed some architectural resource limit.
- For many applications, achieving good performance on parallel accelerators requires deep loop pipelining, which requires dramatically reordering the individual operations in the application. Local dependencies between operations can be respected by compilers relatively easily, but non-local dependencies force implementations to choose between conservatively not reordering operations (which might kill performance), proving that reordering preserves the meaning of the program (which is impossible in the general case), or making unsound transformations (which programmers generally dislike). I propose a mostly sequential operational semantics for C-level streaming languages targeted at parallel accelerators that offers enough flexibility to the implementation to achieve good performance, deviates from conventional program-order semantics in fairly modest and understandable ways, and provides tools with which the programmer can control the reordering performed by the implementation.

These innovations are evaluated in the context of Macah, a new C-like language developed in the Mosaic group at the University of Washington. For validation we use a number of compute-intensive benchmarks developed by members of the Mosaic group and other contributors.

TABLE OF CONTENTS

Page

LIST OF FIGURES

ACKNOWLEDGMENTS

I am grateful to a number of people, without whom I would not have been successful in graduate school.

My advisors Carl Ebeling and Scott Hauck provided an enormous amount of guidance over the years. Most of all, I need to thank them for teaching me that research is about more than coming up with ideas that sound neat. It is also about the careful analysis and experimentation needed to build strong arguments, and being a persuasive writer and speaker. I wish I had realized earlier how much I had to learn about these skills.

My other committee members have also made important contributions. Dan Grossman, in particular, has spent considerable time helping me bring a formal semantics perspective to programming accelerators. Brad Chamberlain contributed an outside perspective and a number of useful observations. Eric Klavins, my graduate school representative, has been diligent in his role ensuring that the process proceeded correctly.

During my time in the Mosaic group, the other members were (in alphabetical order) Allan Carroll, Stephen Friedman, Robin Panda, Brian Van Essen, and Aaron Wood. Without the systems that we developed together I certainly would not have been able to complete my own work. More important than the code were the countless impromptu meetings at the whiteboard to discuss ideas. This dissertation would not be half of what it is without the rich exchange I had with the members of this group. In particular, Brian and I worked closely across a startup company and two graduate schools; it has been a stimulating collaboration, and I wish Brian the best as we now go our own professional ways. Adam Knight and Mikey Levine were not official Mosaic group members, but they both made contributions to the compiler and runtime system that I worked on.

Developing a new language requires benchmark programs, and if the raison d'être of that language is to make something easier, it is critical to get people other than the core

developers to try it out. In addition to the Mosaic group members, I was fortunate to work with many undergraduate and masters students at UW on implementing a number of interesting applications in Macah. The Macah programmers were (again, alphabetical): Abhishek, Danny Anderson, Guy Bordelon, Elliott Conant, Jesse (Randy) Cork, Richard Crouch, Milad Hashemi, Robert Horrox, Jordan Hoyt, Lavanya Jandhyala, Patricia Lee, Brian Mayton, Kristofer Plunkett, Andy Turner, Yuhong Wang, Ben Weintraub, and Ziyuan (Mo) Zhang. Their code helped enrich the Mosaic system, and their feedback helped me develop Macah.

Graduate school would not be possible without the help of staff members who keep an eye on life outside of research, make sure that the administrative gears turn smoothly, and keep the cluster glowing. I am particularly grateful to Kay Beck-Benton, Lee Damon, Shannon Gilmore, and Lindsay Michimoto.

I was financially supported by a number of generous funding agencies during graduate school, including the Department of Energy (grant #DE-FG52-06NA27507), the National Science Foundation (grants #CCF-0426147 and #CCF-0702621), and Microsoft (by way of an endowed fellowship from the Computer Science and Engineering department).

Finally, I want to thank the entire Computer Science and Engineering community at UW; it is special place with a friendly and collaborative atmosphere (in addition to a lot of really smart people, of course).

Chapter 1

THE PARALLEL COPROCESSOR ACCELERATOR ECOSYSTEM

Parallel coprocessor accelerators offer much higher performance and energy efficiency than conventional uniprocessors on applications that fit certain constraints. Many applications of great commercial and/or scientific interest have been successfully accelerated with parallel coprocessors. However, though a variety of parallel coprocessors have been researched, developed and marketed as general purpose applications accelerators, they are still not widely used outside of their native niches.

One of the important reasons for the slow adoption of accelerators for "general purpose" applications is that the programming languages and surrounding tools for them are different from and significantly harder to use than conventional development environments. In fact, there is considerable anecdotal evidence that programming accelerators with their native tools is harder than multithreaded programming with shared memory and locks, which is itself widely regarded as much harder than conventional sequential programming.

In this dissertation I propose and analyze programming language and compiler technologies for making accelerators easier to program. Specifically, these technologies enable "C-level" programming of accelerators. C-level is in quotes because accelerators are sufficiently different from conventional processors that I believe it does not make sense to program accelerators with conventional languages like C and rely on aggressive compiler analyses and optimizations to bridge the large gap between language and architecture. The programming models and technologies proposed in this dissertation make the accelerator programming experience more like the C programming experience, while explicitly acknowledging that accelerators are different from sequential processors in some important ways.

The conventional tools for programming accelerators are hard to use because they do not provide enough abstraction. However, there are costs to abstracting too far from the underlying hardware. This project aims to provide for accelerators the convenience and

portability of C-level programming, without requiring unrealistic compiler technology.

1.1 Parallel coprocessor accelerators

For most of this dissertation the term "parallel coprocessor accelerator" is a broad umbrella that covers field-programmable gate arrays (FPGAs), graphics processing units (GPUs), coarse-grained reconfigurable arrays (CGRAs) [BBKG07, VWC+09], massively parallel processor arrays (MPPAs) [YMA+06, BJW07], systolic arrays and "many-core" processors [SCS+08]. There are important differences between these families of architectures, but they have enough similarities that it makes sense to define an abstract model that covers them all with reasonable fidelity. Such abstract models make it easier for programmers to understand the essential differences between conventional processors and accelerators.

All accelerators are much more computationally dense than conventional processors. Modern single core processors have fewer than 10 execution units and have a maximum instructions per cycle (IPC) of 3 or 4, even when running highly parallelizeable programs.[1] In the same silicon area, accelerators have many hundreds of execution units and have a correspondingly higher peak computational throughput.

Of course, execution units are not free. Compared to conventional processors, accelerators are weak on branch-heavy and unpredictable applications because they lack structures like branch predictors, reorder buffers and sophisticated global caches.

There are also important differences in the memory hierarchy. Accelerators execute many more operations per unit time than conventional processors, but the amount of data that can be moved across a chip boundary is limited by the amount of power it takes to drive huge off-chip wires. Therefore, algorithms running on accelerators cannot perform as many input/output operations per computational operation in aggregate. Often sophisticated buffering schemes are used to get the most out of on-chip memory.

In my work I focused more on FPGA and CGRA style accelerators than GPUs or MPPAs. I believe that the techniques I developed could be extended to other kinds of accelerators, but there is work left to do there. The model and language I propose help blur

[1] SIMD instruction sets, like SSE and AltiVec, allow conventional processors to scale up a bit more. However, processors with these technologies still have far lower parallel execution capacity than accelerators.

the lines between different accelerators, but do not completely eliminate them.

1.2 What accelerators are good for

Researchers have demonstrated that accelerators can achieve much greater performance and energy efficiency than conventional processors in a wide range of application domains including image, audio and video processing, digital communications, encryption, scientific and financial simulations, computer vision, numerical methods, neural networks, and bio-sequence alignment [BGT07, CLS+08, JTLC09, TCK09, THL09, BNW+10, CSJC10].

Hundreds (perhaps thousands) of papers on using accelerators for particular applications have been published in a number of different research communities. The small selection cited here exemplifies a recent trend towards evaluating the same application on more than one family of accelerator; FPGA versus GPU is the most common comparison, though others are sometimes included. This trend signifies an emerging consensus that different kinds of accelerators, which have historically inhabited isolated niches, have quite a lot in common.

Different kinds of accelerators have different strengths and weaknesses, though accelerator comparison studies have not yet painted a complete and consistent picture of what those differences are. At least FPGAs and GPUs will continue to exist as options for application acceleration for the foreseeable future, and it is quite likely that other styles of accelerators will continue to compete for market and mind share. Programming language and compiler technologies that abstract away from architectural details help make comparisons between different kinds of accelerators.

The common characteristics shared by all applications that work well on parallel coprocessor accelerators are:

1. A large amount of work in a small amount of code (repetitiveness).
2. A large number of parallel primitive operations (fine-grain parallelism).
3. Relatively little dynamic decision making (predictability).
4. Relatively simple data access patterns (regularity).

The easiest applications for accelerators are basic linear algebra operations like dense matrix-matrix multiplication, 2D convolution and finite impulse response filters, which are often

referred to as *brute force*. Brute force applications have all the characteristics listed above.

There is another collection of applications—like fast Fourier transforms and dynamic programming—that typically have somewhat more complex control flow and data access patterns and somewhat less available parallelism than brute force applications. I call these *efficient* applications, and they typically work well on accelerators, though supporting them well is more of a challenge for architects and language/compiler designers.

The most complex kind of applications that work on accelerators usually involve algorithmic techniques that improve algorithmic efficiency at the expense of predictability and regularity. Such techniques include application-level caches and data-dependent heuristics. Some applications in this category are heuristic motion estimation for video compression and molecular dynamics simulations that use heuristics to ignore the interactions of distant particles. It requires careful analysis to decide whether an application of this kind can benefit from acceleration on a coprocessor at all. Good language and compiler support is also a greater challenge, compared to the simpler kinds of applications.

Applications whose performance is not dominated by a few small pieces of code are not currently—and likely never will be—good candidates for acceleration on a coprocessor. These kinds of applications include operating systems, office productivity applications, and email or web servers.[2] These application domain restrictions mean that accelerators are useful for a subset of programmers, and the low level of abstraction provided by current tools restrict this subset further.

Interest in coprocessors for general purpose application acceleration seems to have increased in recent years. The slowing of performance scaling for conventional processors has certainly contributed to this trend, as has an increasing interest in energy efficient computing. As an example of this increased interest level, on gpgpu.org—a website devoted to the use of GPUs for "general purpose" applications—there are close to 100 events like workshops, tutorials and conferences listed for 2009 and the first half of 2010. Sustaining this level of interest from programmers outside the core accelerator research communities will require more accessible programming tools.

---

[2]There may be pieces of these larger applications that are amenable to acceleration, such as speech recognition[ZZH+09] or network intrusion detection[GBL10].

1.3 How engineers program accelerators today

Most people doing practical application development today with accelerators use the "native language" of a particular family of accelerators. For FPGAs this means hardware description languages (HDLs) like Verilog and VHDL[3]; for GPUs it means systems like NVIDIA's CUDA[3] and ATI's CTM.[3] HDLs and GPU languages are different in some important ways, but they both force application designers to spend a relatively large amount of time and energy thinking about low level implementation issues.

HDLs were designed for digital circuit design, and HDL programmers still need to think like digital circuit designers. For example, implementation details like pipeline stage balancing and arbitration for shared physical resources are generally left to the programmer. Managing these kinds of issues makes developing for FPGAs something that a large majority of software engineers would never consider doing.

CUDA and CTM are significantly easier to learn than HDLs according to most of the papers that compare GPU and FPGA development. However, these GPU systems still force the programmer to organize their application in a certain data parallel style. Also some implementation details like the exact sizes of the on-chip memories can be issues that the programmer has to think about explicitly.

Designing a brute force application in an accelerator native language certainly requires more work than writing the equivalent in C for a conventional processor, but it is doable. The more complex the application, the more painful the extra implementation detail-level thinking required by accelerator languages is. Thus, providing better programming tools not only makes implementing particular applications easier, but it also opens the doors to applications that are currently considered too complex to accelerate.

1.4 How researchers think engineers should program accelerators

There are many existing programming language and compiler research projects aimed at making it easier to program accelerators. Table 1.1 has an extensive, but not compre-

[3]VHDL, CUDA and CTM are acronyms for (Very High Speed Integrated Circuit) Hardware Description Langauge, Compute Unified Device Architecture, and Close To Metal, respectively.

hensive, list of such projects, along with their approximate year of introduction. Though these projects have made important contributions to language and compiler technology for accelerators, there are still ways in which they do not provide a convenient model for programmers.

Language/Compiler	Target	Company/Organization	Intro
Lucid[AW77]		U of Waterloo/U of Warwick	1977
Handel C[Cel04]	FPGAs	Oxford/Celoxica	1996
Napa C[GS98]	FPGAs	National Semiconductor	1998
RaPiD-C[CFBE98]	RaPiD[CFF+99]	U of Washington	1998
SystemC[Pan01]	Hardware modeling	Open SystemC Initiative	1999
SpecC[GZD+00]	Hardware modeling	UC Irvine	1999
DIL[GSB+00]	PipeRench[GSM+99]	Carnegie Mellon U	1999
Garp compiler[CHW00]	Garp[HW97]	UC Berkeley	2000
Streams C[GSAK00]	FPGAs	Los Alamos National Laboratory	2000
Cynthesizer	FPGAs/ASICs	Forte	2000
StreamC/KernelC [KDK+01, Mat01]	Imagine[ADK+04]	Stanford	2001
MATCH[HNC+01]	FPGAs	Northwestern U	2001
SA-C[BHD+02][4]	FPGAs	Colorado State U	2002
CASH/Pegasus[BG02, VBCG04]	ASICs	Carnegie Mellon U	2002
Sea Cucumber [TJH02]	FPGAs	Brigham Young U	2002
StreamIt[GTK+02, TKA02]	RAW[TKM+02]	MIT	2002
Impulse C[PT05]	FPGAs	Impulse Accelerated Technologies	2003

PICO Express/ Extreme	FPGAs/ASICs	Synfora (acquired by Synopsis)	2003
Cg[MGAK03]	GPUs	NVIDIA	2003
Carte[Poz05]	FPGAs	SRC Computers	2003
Catapult C	FPGAs/ASICs	Mentor Graphics	2004
DIME-C	FPGAs/ASICs	Nallatech	2005
Mitrion C	FPGAs	Mitrionics	2005
Trident[TPA+05]	FPGAs	Los Alamos (mostly)	2005
Accelerator[TPO05]	GPUs	Microsoft Research	2006
AutoPilot	FPGAs/ASICs	AutoESL	2006
CHiMPS[PBD+08]	FPGAs	Xilinx/U of Washington	2008

A complete survey of all the similarities and differences between these projects is beyond the scope of this dissertation. What I present here is a more brief description of the categories they fit in and how my work relates to the most similar ones. The three categories these projects fit in are: (1) compilers for something close to standard C; (2) very abstract languages; (3) C-like languages that are specialized for accelerators.

Compiling Standard C. The project that pushed support for compiling standard C to an accelerator-style target the farthest is CASH/Pegasus. The CASH compiler supports recursive function calls, arbitrary use of pointers, and dynamic memory allocation. While this is certainly an interesting challenge, it misses a very important point. One of the central reasons for the success of C on conventional processors is that it provides a level of abstraction that is considerably more convenient than assembly language, but retaining a substantial amount of control over implementation details like memory allocation and byte-level data structure layout. This is a good level of abstraction for non-accelerator applications like network stacks, file systems and process schedulers. However, C was not designed for accelerators and has some important shortcomings when used for that purpose.

The problem with compiling C to hardware is that accelerators are sufficiently different

[4]Not to be confused with the other Single Assignment C, which is a functional array processing language for multiprocessors.

from conventional processors that it is hard to argue that accelerators are instances of the standard sequential processor model. The gap between the (sequential) model that C was designed for and the fine-grained parallel nature of accelerators means that any effort to get good performance on an accelerator from programs written in standard C requires radical program transformation. Choosing the best transformations to apply in general is an extremely hard problem. On top of that, requiring radical transformation ruins the relatively simple connection between source code and implementation that is one of C's greatest strengths.

Work on automatically vectorizing and parallelizing C compilers for supercomputers is in many ways closely related. Analyses of automatic vectorizing compilers [AJ88, LCD91, Smi91] illustrate well the strengths and weaknesses of this approach. Though it is certainly possible to vectorize some programs written in standard C automatically, there are non-trivial gaps between the kinds of loops that can be automatically vectorized and the kinds of loops that can be hand vectorized by a human. These gaps seem inevitable and represent an important software engineering weakness: there are applications that in principle can be vectorized, but cannot be programmed with an automatic vectorizing compiler, because they do not fit the recognized patterns. Modern accelerators, like FPGAs and GPUs are substantially more flexible architectures than conventional vector processors, which makes it even harder to compile to them from sequential code automatically.

There is a marketing issue that inevitably comes up in the context of "standard C" compilers for accelerators. Some of these compilers—especially the more commercial projects—claim to accept standard C, but stretch the definition of "standard C" badly. These kinds of projects generally do not propose any syntactic extensions to the language, but require programs to be written in a very restricted style with many calls to special intrinsic functions and/or pragma directives. Such restrictions are essentially a separate target-specific language embedded in C. This is not a bad technical direction, but it is important to recognize projects in this category (for example, Impulse-C) as C-like languages, rather than compilers for standard C.

Abstract Languages. A number of research languages for accelerators (and supercomputers) appear to have started with some version of the thought "Programmers like

sequential languages like C, but typical C programs have too many implementation decisions coded into them, so let's define a language in which algorithms can be expressed more abstractly." The most common direction in which this thought leads is towards data-parallel languages like Accelerator and ZPL[CCL+98].

Data-parallel languages have two important problems for compilation to accelerators. First, though they avoid issues like alias analysis that can be very problematic when compiling from standard C, there are still hard implementation choices to make. Automatically choosing what kind of buffering and loop optimization strategies to use is hard. The second problem is that not all algorithms can be expressed naturally in a data-parallel style. Complex applications often involve data structures and control flow patterns that dramatically improve efficiency, but cannot be expressed easily in abstract data-parallel terms. An interesting example of this tension can be seen in the development of Chapel[CCZ07], which was heavily influenced by the project ZPL. Relative to ZPL, Chapel is a much less purely data-parallel language, in no small part because the developers found the need for "harder" data structures like graphs and hash tables.

In the interest of fairness and completeness, the high level of abstraction of languages in this category can make aggressive automatic optimization significantly easier. Examples include loop fusion and array contraction in ZPL [LLS98] and dataflow operator fusion in StreamIt [ATA05].

A related but different approach to programming accelerators is the library method, where we assume that a small number of "gurus" will write a modest number of core routines that can then be stitched together by a larger number of less skilled programmers. This may thus open accelerators to a much larger audience without exposing them to the internal details of accelerator programming.

The library method can be useful for programming within specific application domains. However, it does have two important weaknesses. First, the libraries have to be implemented in some language, and the harder it is to use that language, the smaller the pool of library-writing gurus will be. Second, for application developers there is an extremely steep learning curve if they want to implement functionality that is not provided by available libraries.

"C-Like" Languages. Given the significant challenges that remain in the previous two

categories, I believe the greatest promise for improving the programmability of accelerators for the foreseeable future lies with C-like languages. "C-like" means languages that strike the same kind of balance between abstraction and programmer control as C, but in the context of accelerators instead of sequential processors. I refer to C here both admiringly and derisively as a "portable assembly language"; this is exactly the level of abstraction that I have in mind; just enough that well-written programs can be efficiently compiled to a range of related architectures.

The majority of the projects listed in Table 1.1, as well as the language that I developed as a testbed for the contributions described in this dissertation, fit in this category. It is interesting to examine the similarities among these languages. These commonalities match nicely the features of the abstract accelerator architecture model described in Chapter 2, even though we designed the model to be an abstraction of architectures, not to match these languages.

- Kernels. There is significant overhead involved in configuring an accelerator to run a piece of code, which means that an application has to spend a large amount of time in a particular block of code in order to amortize this overhead. These blocks are typically called kernels, and most C-like languages leave kernel identification to the programmer. In some cases the semantics are subtly different inside of kernel blocks, either by definition or implicitly by how kernels are implemented.
- Parallelism. The performance advantages of accelerators come from running hundreds to thousands of primitive operations simultaneously, so clearly the programmer and/or compiler have to identify where such parallelism can be found in a program. All C-like languages for accelerators include either explicit parallel looping constructs or hints of some kind that indicate where loop optimizations should be applied. Note that thread-based parallelism is uncommon in languages designed for FPGAs and closely related architectures. Languages designed for GPUs often include thread-like parallel constructs, but in highly restricted forms.
- Data handling. C-like languages for accelerators all include special handling of data. The two main issues that involve data are: (1) Accelerators have large on-chip

buffers that are mostly or entirely software-controlled. (2) The large amount of semi-automatic parallelization that is usually done makes accessing main memory through unrestricted pointers a major challenge. Many C-like languages for accelerators force the programmer to choose explicitly what data structures should be allocated into local memories. Also, non-pointer methods for accessing main memory, like streams, are common.

- Tuning. Putting implementation decisions like local data buffering and loop optimizations partially or entirely in the programmer's hands forces the programmer to make choices like how large buffers should be and how much particular loops should be unrolled. These tuning decisions depend directly on the resources available in a particular target architecture, but one of the major goals of C-like language design is to get away from encoding architecture-specific decisions in source code. This means that support for automatic or semi-automatic tuning is an important issue.

These are the most important ways in which C-like language for accelerators are different from C, but there are others, like optimization of the number of bits used to represent numbers [BSWG00, SBA00] and graceful handling of exceptional conditions [TKS+05]. My work is focused on improving language and compiler support for the core issues.

1.5 Contributions of this dissertation

To improve the programmability of accelerators I developed an abstract model and a number of programming language and compiler technologies. I did this work in the context of the Macah language that my colleagues and I in the Mosaic group at the University of Washington developed as a testbed for accelerator programming research. This dissertation makes contributions in four distinct areas of accelerator programmability:

- Abstract architectural models like the von Neumann model give programmers an idea of the resources and performance characteristics of a family of computers. Unfortunately, the von Neumann model does not faithfully represent accelerator architectures. In Chapter 2 I define a model that we developed specifically for accelerators. I also

demonstrate how the model can be used for high-level performance analysis of algorithms.

The Macah language and the Mosaic toolchain are described in Chapter 3. There I describe the connections between Macah, the abstract model and the benchmarks we developed. I also sketch the most important parts of a compilation flow for Macah, and by extension other C-like languages for accelerators.

- In Chapter 4 I define and analyze enhanced loop flattening, a new loop optimization framework that allows conventional loop pipelining algorithms to be applied to program sections with arbitrary static control flow. Loop pipelining is an important transformation for compiling C-like languages to accelerators, because it allows independent operations from different iterations of a loop to execute concurrently. Conventional loop pipelining algorithms are applicable only to inner loops, which can create significant inefficiency in the prologue and epilogue sections of inner loops in more complex loop nests. Loop flattening allows prologue and epilogues of adjacent loops to be overlapped, and enhanced loop flattening allows iteration distances between specific program points to be controlled more precisely, which can improve performance.

  From a software engineering perspective, the important consequence of enhanced loop flattening is that programmers can get the benefits of pipelining for more complex applications without manually reorganizing the code in a much less natural style.

- In Chapter 6 I propose a system for programmer-guided tuning of Macah programs to specific architectures. The system uses "tuning knobs" explicitly declared by the programmer and typically used for things like buffer sizes and loop bounds. The system performs an automatic empirical search for good values for the tuning knobs in a program. The most important novelty in this tuning system is that it simultaneously optimizes some quality function and satisfies resource constraints. Previous approaches to tuning just optimized a quality function and assumed that all possible configurations of the system actually work.

  Incorporating constraint satisfaction into tuning is particularly important for accelerators because they have many resources like distributed local memory and on-chip

networks for which there are no automatic fallback mechanisms if a program tries to overuse the resource. A diverse set of approaches to tuning have been proposed; I present a brief survey in Chapter 5 to help contextualize my own work.

- Several C-like languages have a combination of features that interact in complex and problematic ways: static reordering of code (for example, by loop pipelining) and streaming I/O. Static reordering is generally done in a way that respects dependencies through local variables,[5] but not through stream sends and receives. For extremely simple streaming communication patterns, reordering that respects local dependencies preserves global program behavior. However, cycles in the stream communication graph, and especially stream buffers with finite bounds, introduce serious semantic problems.

  In Chapter 7 I develop operational semantics for a core subset of Macah and give examples of programs for which common practices in C-like language implementation can cause incorrect behavior. I also prove that these common practices are safe for specific subsets of programs and suggest how these findings could be used to refine the definition of C-like streaming languages and debugging tools for such languages.

Finally, in Chapter 8 I summarize my contributions to programming accelerators with C-like languages and discuss the most important directions for future work.

[5] Dependencies through pointers and arrays are an important challenge for which partial solutions exist. I largely work around this issue in Macah by insisting that all global data transfers go through streams.

Chapter 2

AN ABSTRACT MODEL FOR PARALLEL COPROCESSOR ACCELERATORS[1]

In [Sny86], Snyder argued eloquently for the importance of computing models[2] that are a "region of consensus, ... explicit about a few salient features [of a family of computers] and mute on everything else". Architectural models define, in the most abstract terms possible, the resources, behavior, and performance characteristics a programmer can rely on from any conforming computer, and what a compiler writers and architects are obliged to provide in one form or another. Performance characteristics of the hardware are defined by costs associated with operations within the model. Successful models can serve as the central archetype for a variety of computers, languages and algorithms. However, models that do not paint a realistic picture, such as the PRAM model for multiprocessor computers, can lead to unrealistic expectations on the part of algorithm designers, and thus theoretically optimal algorithms that are in no sense optimal on any realistic machine[GMR99]. Models are contracts between algorithm designers, language designers and architects working within a broad family of computers in the same way that instruction set architectures (ISAs) are contracts between programmers, compiler writers and micro-architects for a specific line of processors.

As background on the meaning and use of models, we will examine the von Neumann model. Then we delve into the definition of a new model for accelerators, how it connects to actual families of accelerators and how algorithm designers can use it to get first-order performance estimates.

---

[1]This work was originally published in [YVE06].

[2]The original paper on the material in this chapter used Snyder's term "type architecture" for "model of a family of architectures". We have stopped using that term because many people found the use of the word "type" confusing.

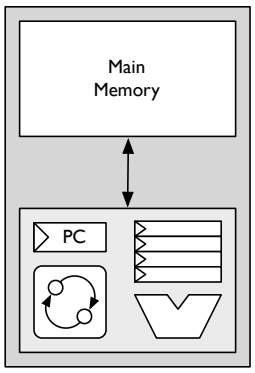


Figure 2.1: An illustration of the components of the von Neumann model.

2.0.1 The von Neumann model

By far the most well-known compute model is the von Neumann machine. The components of the von Neumann machine are a large, random access memory and a processor consisting of a functional unit that can compute some set of simple functions, a small amount of state (often referred to as a program counter or PC) and a controller that orchestrates the operation of the machine. These components are illustrated in Figure 2.1.

A von Neumann machine runs a simple fetch, execute, and store loop. The only cost in the von Neumann model is that each instruction execution (or memory reference) carries unit cost.

Implementations of these kinds of models are necessarily approximations. For example, implementations of the von Neumann model generally use registers and caches to provide the appearance of unit-time memory access. Most programmers only pay close attention to the exact details of a particular computer's memory system when extreme optimization is called for.

An implementation of a model is also free to behave in ways that might seem to violate the model, as long as the essential interfaces are maintained. For example, modern out-of-order processors maintain the appearance of executing instructions in the conventional "von Neumann" order, while internally the instruction execution is reordered. Maintaining the appearance of strictly in-order execution is a constant concern for architects. For example,

many clever architectural innovations for speeding up the execution of programs have been proposed and not implemented because they made it too hard to maintain precise interrupts.

C (and many other sequential languages) reflect perfectly the von Neumann model. The language assumes one large pool of memory which the programmer is free to manage, and a sequential one-expression-at-a-time evaluation strategy. This close connection between language and model means that even for relatively low level systems programming, programmers can write generic C with a generic processor in mind. Only for a few specific kinds of programming is it necessary to know specific hardware configuration information.

Compiling plain "dusty deck" C code to parallel accelerators is hard for a number of technical reasons; the overarching issue, though, is that accelerators are simply not instances of the von Neumann model. To help visualize the disconnect, imagine a delivery service in a city. Conventional processors are like small cars that can pick up small loads and maneuver around the city well. Accelerators are like tractor trailers, potentially much more efficient for hauling big loads, but not appropriate for other kinds of delivery jobs. Compiling C to accelerators is like taking a delivery plan designed for a small car and using it with a tractor trailer instead. In the best case, it might be possible to adjust the plan automatically to work well for the bigger vehicle, but doing so can be quite hard. And in many cases, such adjustment is not possible at all.

2.1 A proposed model

In order to define an abstract model for parallel coprocessor accelerators, it is necessary to analyze the common characteristics of actual accelerators. First, something that is embedded right in the name **co**processor is that accelerators are complementary to general purpose processors. This means that the model should be a hybrid, with a conventional von Neumann part for everything except the kernels, and an accelerator part for the kernels.

The defining architectural features of accelerators themselves are:

- a large number of simple, concurrent, densely packed compute units
- a distributed local memory hierarchy
- a scalable local communication mechanism

- simple and efficient control

Connections between these abstract features and particular architectures are covered in more detail in Section 2.2.

There is an important distinction between fine-grained parallelism, which accelerators can exploit,[3] and coarse-grained parallelism (task-, process-, or thread-level). Algorithms that can be implemented well on an accelerator can, in many cases, be implemented on multiprocessor systems in a task-parallel style as well. However, when looking at the cost per operation, in terms of both dollars and energy per operation, multiprocessor architectures have more overhead than accelerators. Thus, the close, fine-grained communication between the operations in some computations that work on accelerators precludes an efficient implementation in a task-parallel architecture like a multiprocessor. Many computations exhibit both task- and accelerator-parallelism and for those it is reasonable to build a multiprocessor with accelerator nodes. Examples of such machines include the Cray XD1 [Cra], SCORE [CCH+00], and Merrimac [DLD+03].

2.1.1 Hybrid accelerator model

I propose the hybrid accelerator model (HMP)[4] as a tool to improve the accessibility of coprocessor accelerators. The HMP model is an extension of the sequential von Neumann machine and describes a variety of computers, from FPGAs with embedded sequential processors, to hybrid reconfigurable computers based on architectures like PipeRench [GSB+00] and RaPiD [CFF+99], and to some degree SIMD architectures like Imagine [KRD+03] and vector processors. The components of the HMP model are illustrated in Figure 2.2. Execution is performed by two distinct components: a sequential, von Neumann style, processor on the left and an accelerator on the right. The hybrid architecture executes a single program with a single thread of control; the locus of execution can switch from the sequential processor to the accelerator and back, based on the kind of computation currently being executed. Abstractly, these two execution engines share all memory resources.

[3] "Accelerator parallelism" includes instruction-level, loop-level, and data-level parallelism.

[4] HMP was an acronym for "hybrid micro-parallel", which is also a term from [YVE06] that we do not generally use anymore. For the sake of consistency we continue to use the HMP acronym.

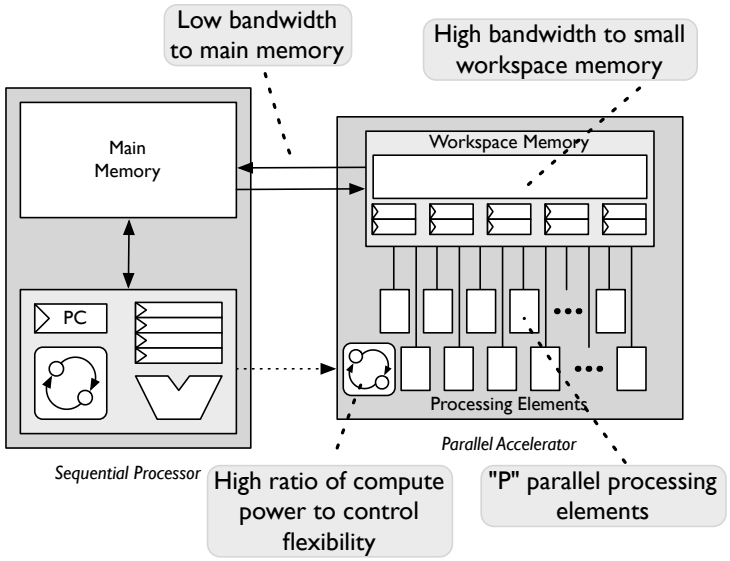


Figure 2.2: "HMP" model. The left side is a conventional von Neumann machine and the right side represents the accelerator proper.

The accelerator contains a "workspace" memory, an array of P functional units, and control resources of some sort. The specific number, type, and organization of the functional units are unspecified and vary from implementation to implementation. As a rough guideline, programmers should expect tens to hundreds of functional units. In principle the number of functional units could be scaled up much higher. In practice architectures with many thousands of functional units usually have additional levels of hierarchy that make programming the whole system as a single accelerator challenging.

The workspace memory models all the registers and memories distributed throughout the accelerator. The size of workspace memory is implementation-dependent, but is *much* smaller than the main memory. As a rough guideline, the workspace memory is around the size of an L1 or L2 cache on an equivalently sized sequential processor. An example of workspace memory are the embedded RAMs and registers distributed throughout the fabric of an FPGA. A key aspect of the model is that the workspace memory and the bandwidth between workspace memory and the functional units should be sufficiently large to keep all the functional units continuously operating. That is, the internal data bandwidth within the

accelerator is high enough to sustain maximum parallelism, while the bandwidth between the main memory and the accelerator is limited in the same way that this bandwidth is limited in the sequential processor. The communication between main memory and the workspace memory is, in many cases, specialized to support high bandwidth for typical memory access patterns. For example, programmable direct memory access (DMA) channels are a simple way to support the predictable memory accesses made by many signal and media processing algorithms.

The workspace memory of the model explicitly exposes one of the major challenges encountered in programming an HMP computer: that of scheduling the computation so that the required data is in the workspace memory when it is needed. Note that automatically managed caches, commonly found in sequential processors, are not present in most accelerators, and are generally not an efficient way to exploit the simple, predictable access patterns that exist in most accelerator-friendly algorithms. Caches can handle simple and predictable memory access patterns well, but manually managed memories can do so as well, with less circuitry and energy.

In addition to its conventional link to main memory, the sequential processor also has a link to the workspace memory that is used to maintain shared state. This is the secondary role of the low bandwidth/high latency link at the top of the figure. Finally, there is a control link between the sequential processor and the accelerator to indicate that their execution is coordinated.

Execution model

The rules governing the execution of the HMP model are an extension of the rules governing the von Neumann machine. When a program begins executing, it runs on the basic von Neumann machine. At some point during execution, the machine can transition from sequential mode to accelerator mode. In accelerator mode, the sequential processor is inactive and the controller in the accelerator orchestrates execution. While in accelerator mode, any number of the functional units and the links between the accelerator and the workspace memory may be active simultaneously in a single execution step. When the accelerator

finishes its task, control transfers back to the sequential processor. The accelerator cannot transfer control back to the sequential processor in any way other than ending its current task–either normally or as a result of some exceptional condition.

Control resources in accelerators are limited and often optimized for algorithms that perform the same operation or group of operations many times in some repeating pattern. This fact limits both the sophistication of the control flow and the data access patterns that are supported well. Unfortunately, the diversity of real accelerators makes it impossible to describe precisely the flexibility of the controller in the abstract model. However, a clear theme is evident: regularity and predictability are important in both data access patterns and control flow. Predictability is important because high performance accelerators are deeply pipelined and, as in any computational pipeline, unpredictable changes in control flow can cause large amounts of work to be discarded in later stages of the pipeline. Furthermore, accelerators are unlikely to provide resources for techniques such as branch prediction that help mitigate the impact of unpredictable control flow. Regularity is important because accelerators have limited means to coordinate and control the independent operation of large numbers of functional units. An algorithm need not be perfectly predictable and regular to run on an accelerator, but the more predictable and regular the algorithm, the more likely it is to fit within the constraints of a given architecture and use the available resources efficiently.

Performance model

The cost model of the HMP model is more complicated than that of the von Neumann machine but inherits the standard unit cost of executing an instruction on the sequential processor. In addition, there are two significant costs for transitioning from sequential to accelerator mode. The first cost, T_C, models the accelerator's configuration time, and is at least two orders of magnitude greater than the standard unit cost. The second cost, T_I, models the startup and initialization time, and can be anywhere from one to two orders of magnitude greater than the standard unit cost. The reason these costs are separate is that most accelerators can cache at least one configuration, and therefore the configuration

cost–but not the startup cost–can be avoided if a program enters a particular kernel several times. Both of these costs encourage programmers to make a transition to accelerator mode only if there is sufficient work to do.

Once in accelerator mode, the cost of executing up to P operations simultaneously, given an accelerator with P functional units, is some small factor, α, times the standard unit cost. This α factor, which typically ranges from 1 to 10, models the fact that sequential processors typically can execute a single instruction faster than a comparable accelerator. The cost does not vary with the number of operations executed, so it is clearly beneficial to execute as many operations as possible. The crucial limitations on the operations executed in a single step are that none of them can depend on any other, since they are executing simultaneously, and all but a very small number must fetch their operands from the workspace memory, since the bandwidth to main memory is so low. There are two immediate and important consequences of these costs: To achieve full utilization, it must be possible to amortize each piece of data transfered between main memory and workspace memory over many operations, roughly proportional to P/β, where β is the bandwidth between the main and workspace memories. Also, while operating in accelerator mode, each kernel's current "working set", for a given algorithm, should entirely or largely fit in the workspace memory. If the working set does not fit, then the achievable parallelism is limited by the low-bandwidth connection to main memory, thus reducing or eliminating any potential for increased performance.

2.2 Implementations of the HMP model

Implementations of the HMP model are free to let the sequential processor and accelerator run simultaneously, either overlapping sequential and accelerated portions of a single program, or concurrently scheduling different threads or processes on the two resources, as long as the appearance of non-overlapping execution is maintained. The details of how such concurrent execution is accomplished are implementation issues that programmers should be able to ignore safely.

Shared memory is another area where an implementation of the HMP model may choose to optimize. Although the HMP model dictates that all memory resources are visible to the sequential processor and accelerator, accelerators often have registers and memories that

are not easily accessible from outside and may use data copying and caching to simplify data access. Additionally, static and dynamic program analysis may be used to optimize away data movement or copies of data.

In the remainder of this section we analyze how well several architectures fit the abstractions of the HMP model. This analysis helps clarify the differences and similarities between architectures.

FPGA Platforms It is clear that FPGAs paired with sequential processors can implement the HMP model. Using the HMP model to model FPGA platforms constrains how the spatial fabric is used. For example in the model, the fabric is not used to implement large numbers of independent concurrent "hardware threads", but as a unified accelerator. In return for this restriction, the programmer is able to reason abstractly about how to structure the program so that it will run efficiently on the combined sequential/accelerator platform.

Garp An early example of an FPGA-based hybrid sequential/accelerator architecture that fits the HMP model is Garp [CHW00]. Garp had a single thread of control, an FPGA fabric that was optimized for computation, and specialized memory access units that facilitated data movement between memory and the FPGA fabric. The small size of Garp's workspace memory, which restricts the range of algorithms that can be accelerated, is a constraint that the HMP explicitly models.

Coarse-grained Configurable Research in coarse-grained configurable computing has produced accelerators such as RaPiD [CFF+99] and PipeRench [GSB+00]. When coupled with sequential processors to form hybrid systems, as seen in HASTE [LS03], they fit the HMP model. The RaPiD architecture fits the accelerator in the HMP model well: The workspace memory comprises the datapath registers and embedded memories as well as pipeline registers in the interconnect. Access to memory occurs via specialized memory streams that can support a small number of memory accesses per cycle. Experiments showed that a large number of different algorithms could be executed efficiently on RaPiD because of the large workspace memory. Coarse-grained architectures have some potential advantages over FPGA-based HMP systems: The overhead of switching from sequential to accelerator mode is lower because there is far less configuration data, and cost and power

are reduced because of the custom functional units.

Extended Datapaths Other work, such as the Stretch S5 engine [Wan04] and ADRES[MVV+03], extend a conventional processor by integrating an accelerator into the processor's datapath. Data movement between main memory and the workspace is typically provided by the processor, via wide load/store operations, rather than a dedicated stream engine or DMA engine. These architectures fit the HMP model and are characterized by a low cost to transition between sequential and accelerator modes, small accelerators, and small workspaces with shared links to main memory.

Boundaries of the HMP Clearly not all parallel architectures fit the HMP model well. For example, SIMD architectures such as Imagine [KRD+03] or vector extension units have some features of HMP and not others. Although SIMD architectures have many functional units operating in parallel, they all execute the same instruction stream, and the communication between them is limited.

2.3 Algorithm analysis and design

The most important role of a model is in performance analysis of algorithms. In the case of the von Neumann machine, the analysis is so simple and widely used that we tend to lose sight of the model itself. However, when programmers assume that each arithmetic operation, switch statement, or array access has unit cost, and the total cost of a program is simply the sum of all the costs of its pieces, they are using the von Neumann model.

In this section, we consider several applications and use the HMP model to analyze their performance. Note that this analysis has similarities to that typically done by skilled hardware designers when moving algorithms into hardware.

The speedup achieved using an accelerator depends, of course, on how much of the application run time is spent in code that can be accelerated ($T_{\mu p}$) and what the average achievable parallelism is in the accelerated parts (P_{avg}). We will focus on the latter analysis which determines the parallelism that can be achieved within the constraints of the model. Given this analysis, the speedup for the application as a whole is a straightforward application of Amdahl's law. Given P, α, T_C and T_I, as defined in Section 2.1.1 of the HMP

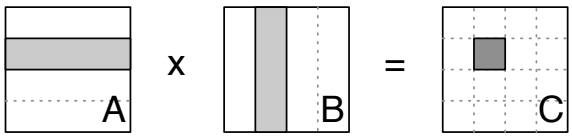


Figure 2.3: Tiled Matrix Multiplication

model, the overall resulting speedup is

$$\frac{T_{seq} + T_{\mu p}}{T_{seq} + T_C + n \times T_I + (T_{\mu p}/P_{\mathit{eff}})}$$

where $P_{\mathit{eff}} = min(P_{avg}, P)/\alpha$ is the effective parallelism relative to sequential execution, and n is the number of sequential/acclerator mode transitions.

The algorithms analyzed here are drawn from MiBench [GRE+01] and a few other sources. The results are summarized in Table 2.1. The algorithms we chose are not novel, nor are their accelerator implementations. Rather they highlight key issues in using the HMP model as an analysis and design tool. In fact, not all of the examples are highly conducive to acceleration, and for those that are not we can use the HMP model to understand what inhibits greater parallelism. The importance of having a model like this is that we can do this sort of algorithm analysis without thinking about the details of any particular architecture.

All algorithms that can be accelerated have at their core a small number of loops. A critical issue in determining the amount of acceleration available in an algorithm is inter-iteration feedback dependencies. If there are no inter-iteration dependencies, also known as loop-carried dependencies, then loop iterations can be executed independently and initiated as fast as resources allow. However, loop-carried dependencies, when they do exist, constrain the initiation rate (the number of loop iterations that can start per unit time) and thus may affect the achievable parallelism. This rate is conventionally call the *initiation interval* of a loop [Rau94b]. Though programmers must be aware of the importance of inter-iteration dependencies, they need not compute the exact initiation intervals of their programs; algorithms for analyzing initiation intervals are known, and compilers should provide clear feedback when acceleration is limited by loop-carried dependencies.

Dense matrix-matrix multiplication, computing the value of `C`, given `A` $\times$ `B` $=$ `C`

Table 2.1: Summary characteristics that make algorithms more or less amenable to acceleration

Kernel (Application)	Params	Bandwidth (units per cycle)	Approximate Workspace	Initiation Interval	Parallelism (P_{avg})	Predictability	Regularity
matrix multiplication	T	wordsize$\times P/T$	wordsize$\times 2\times T^2$	1	T^2	High	High
color conversion (JPEG)	n	$48n$ bits	$8n$ kBytes	1	$15n$	High	High
motion estimation (MPEG)	-	minimal	1+ kBytes	1-6	10-64	Low	Low
2D convolution (image proc.)	k, n	2 pixels	$k^2+(k-1)n$ pixels	1	$2k^2-1$	High	Med
Smith-Waterman	T	64 bits	T kBytes	3	$7T$	High	Med
$\text{Rijndael}_{\text{baseline}}$ (AES)	-	256/14 bits	4 kBytes	14	9.6	High	High
$\text{Rijndael}_{\text{FF\_TABLE}}$ (AES)	-	256/5 bits	32 kBytes	5	9.6	High	High
ADPCM (Audio encoding)	-	24 bits	minimal	14	3	High	High

and values for A and B, is a simple algorithm that offers abundant parallelism. Assuming A, B and C are $M \times N$, $N \times O$ and $M \times O$ matrices respectively, there are MNO multiplications and additions to compute, there are no dependencies at all among the multiplications and the additions break down into MO summations, each composed of N additions. On an HMP computer, however, we must schedule the computation carefully to use the compute, communication and storage resources efficiently.

For large values of N, M and O, the arrays will not fit in the workspace memory. Due to the limited main memory bandwidth, we must stage the computation so that values from the A and B matrices are reused many times whenever they are read into the workspace. One way to accommodate both of these constraints is to break the C matrix into $T \times T$ *tiles*, and to compute each one completely, before moving to the next tile. This well-known Summa algorithm [vW97] is illustrated in Figure 2.3 and the following pseudocode.

```
for (i from 0 to M-1, step by T) {
  for (j from 0 to O-1, step by T) {
    for (x from 0 to T-1)
      for (y from 0 to T-1)
        C[i+x,j+y] = 0;
    for (k from 0 to N-1)
      for (x from 0 to T-1) // Unroll all
        for (y from 0 to T-1) // Unroll all
          C[i+x,j+y] += A[i+x,k] * B[k,j+y];
} }
```

Notice that the the inner two loops are intended to be fully unrolled, which means that an iteration of the k loop computes one multiply-accumulate for each cell of the current tile and can be completed in a single step. Clearly, not all accelerators will have T^2 multiply-accumulators available, but we make the assumption that as long as T^2 is a reasonably small multiple of P, the system can time-multiplex these independent operations efficiently. Note that in order to carry out these T^2 operations, we need T values from the A matrix and T values from the B matrix, and each value is reused T times.

This organization of the algorithm fits within the constraints of the HMP model. First, the amount of state in workspace memory is only proportional to T^2. Second, the ra-

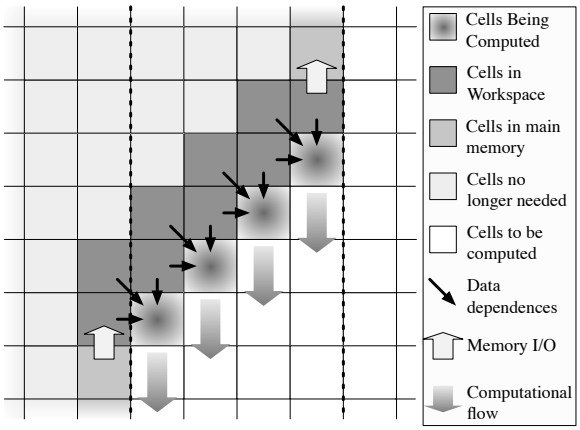


Figure 2.4: Smith-Waterman example with $T = 4$

tio of computation to communication is proportional to T. A single iteration of the `k` loop executes T^2 operations and reads $2T$ values from the `A` and `B` matrices for a ratio of multiply-accumulates to communication of $T^2/2T{=}T/2$. For example, if the memory interface averages only 1 read per cycle and the accelerator can sustain 20 multiply-accumulates per cycle, then a tile size of $T \geq 40$ yields full utilization.

There is some overhead associated with transitioning from one tile to the next, which requires some control as well as additional workspace memory for double-buffering the results. However, with sufficient workspace memory and control resources, the full P parallelism potential of the accelerator is achievable.

Smith-Waterman is a *sequence similarity* algorithm used widely in bioinformatics to search DNA, RNA and protein databases. The algorithm finds highly similar subsequences of two longer sequences, usually called the database string and the query string. This uses a dynamic programming algorithm to fill in a scoring table that has the database string along the top and the query string along the left. The score at each entry of the table indicates how well the substrings of the database and query strings ending at this location match. The computation of each entry in the table requires 21 additions, subtractions and comparisons, and one look-up into a character comparison table. This computation depends on the entries to the top, left, and top-left as shown in Figure 2.4.

Though there are data dependencies between entries, we can find a great deal of parallelism by computing diagonally across the table. Unfortunately, for large genomes this strategy would overwhelm any realistic workspace, because all of the entries along the entire diagonal of the table would have to be resident simultaneously. One solution, illustrated in Figure 2.4, is to break the table into vertical stripes, T columns wide, and compute in a diagonal fashion down each stripe, one stripe at a time. Using this strategy, the average parallelism is $21 \times T/II$ where II is the initiation interval. The amount of workspace storage needed for intermediate results is only proportional to T. Note that this solution requires additional memory bandwidth to save the right-most column of the stripe, which must be read back in when computing the next stripe. However, this requires only one extra read and write, which is amortized over the computation of one row of the stripe. The initiation interval is 3 and thus the average parallelism is $7T$.

The workspace memory must also accommodate the character comparison table. In order to maintain this level of parallelism, the accelerator must execute T/II lookups into the table per cycle, which means that the table must be capable of multiple simultaneous lookups, or that multiple copies of the table must be resident in workspace memory. For protein databases, the table is about 2KB, and in this case the achievable parallelism may be constrained by the size of workspace memory needed for the lookup tables.

ADPCM, which stands for adaptive differential pulse code modulation, is a technique for encoding audio signals that produces lower bit rates than conventional PCM. The core of the ADPCM encoding algorithm is a relatively simple loop that seems to offer a great deal of parallelism. In the MiBench implementation, there are slightly more than 40 operations in the inner loop, and minimal communication is necessary between the accelerator and main memory or the sequential processor. Unfortunately, the initiation interval of this implementation is 14, which means that the average parallelism is only approximately 3 operations ($40/14 \approx 3$). This analysis shows us that there is a clear limit on the benefit of acceleration for this algorithm, barring deeper algorithmic transformations.

The parallelism of this application can be improved if the encoding of multiple independent audio streams can be interleaved. This technique, called C-slowing [WMPW03], can often be applied when loop-carried dependencies constrain the achievable performance

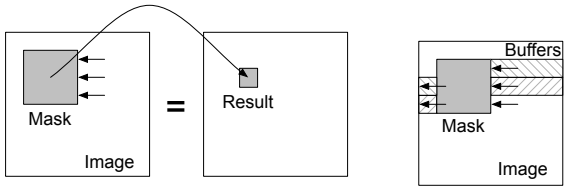
and leverages the ability of the hardware to time-multiplex the same operations between multiple data. By reformulating the algorithm as in [PKCD05], it is possible to achieve even greater parallelism. The main problem addressed in that paper is that saturating accumulations are a bottleneck to high degrees of pipeline parallelism, because they require a tight inter-iteration dependency. By refactoring saturating accumulation to make it more like normal accumulation, it is possible to loosen the inter-iteration dependency considerably. This is a perfect example of the kind of concerns that are important because of the kind of parallelism accelerators exploit, not the details of any particular accelerator.

Color conversion, from RGB to YCC color spaces, as implemented in the MiBench JPEG benchmark, also appears to be amenable to acceleration. The core of the algorithm is a doubly-nested loop with 9 multiplications by a constant, 6 additions and 3 shifts. The multiplications are implemented by table lookups, which may or may not be a good design choice in an HMP computer, depending on the size of the workspace relative to the available multiplication resources. For 24-bit pixels, the total size of the tables is 8 kilobytes, and for 36-bit pixels the total size is 128 kilobytes.

The most serious problem with color conversion is memory bandwidth. Each iteration of the inner loop reads and writes a pixel. Given the restricted bandwidth to main memory in the HMP model, color conversion cannot fully utilize the computational resources of a well balanced accelerator. In some cases, it is possible to mitigate the problem of a low computation to communication ratio by performing two or more algorithms together in the accelerator. For example, color conversion may be part of a larger graphics pipeline, in which case it could be combined with another phase without creating any additional bandwidth requirements.

Motion estimation is a computationally significant part of MPEG-2 video encoding. Motion estimation is used to calculate differences between adjacent video frames, which require less data to encode than the frames themselves. Each 16×16 block of pixels is compared with blocks in its neighborhood in the adjacent frame to find the least different block.

The amount of computation that is needed to do *full* motion estimation is enormous, and many ASIC and FPGA implementations opt to do full motion estimation. However,



(a) k rd / 1 wr (b) 1 rd / 1 wr

Figure 2.5: Communication/workspace tradeoffs for (a) simple and (b) buffered 2D convolution.

the Mediabench implementation uses two techniques to minimize this computation for a sequential processor. First, block comparisons are aborted as soon as the difference exceeds the best difference found thus far. Second, blocks are searched in an order that spirals out from the location of the reference under the assumption that most image movement is small. While these optimizations make perfect sense on a sequential processor, they reduce the predictability and regularity that accelerators depend on.

These two optimizations highlight a weakness in the HMP model: we know that accelerating algorithms requires some predictability and regularity, but real accelerators have a wide range of control mechanisms and internal interconnect networks that may or may not be flexible enough to support optimized MPEG-2 motion estimation. Pure model-based analysis cannot tell us if these optimizations are possible on particular accelerators. It may be necessary to use an unoptimized motion estimation algorithm that performs more computation in return for a regular and predictable algorithm.

Even with the Mediabench optimizations there is a great deal of parallelism ($\approx$64 ops) to exploit in motion estimation. Assuming that the early termination optimization is used, the initiation interval of the inner loop is not a simple number. If that optimization were not present, the initiation interval would be 1. However, if we force each iteration of the inner loop to complete before initiating the next one, in order to check against the best difference observed so far, then the initiation interval is 6. If we optimistically initiate iterations as soon as possible, and cancel them if necessary, the effective initiation interval will be between 1 and 6, and the parallelism available, will be between 10($\approx$64/6) and 64.

2D Convolution is the core of many image processing operations such as noise reduction, image smoothing, and edge detection. 2D convolution consists of repeatedly applying a relatively small convolution kernel to a comparatively large source image, as shown in Figure 2.5. Image processing kernels commonly range from 3×3 to 17×17, for Sobel operators and Laplacian of Gaussian kernels respectively.

In our analysis, we will apply a $k \times k$ convolution kernel to an $n \times n$ source image, where $k \ll n$. To compute a new image, the convolution kernel is applied n^2 times, each comprising k^2 independent multiplies followed by $k^2 - 1$ additions. Since the kernel applications are independent, the initiation interval is 1. If the kernel is applied in row major order, then most of the data can be reused and only k pixels need to be read. This yields a computation/memory bandwidth ratio of about k. Given sufficient workspace memory we can further reduce the communication requirements to reading 1 pixel and writing 1 pixel by buffering the pixels across rows as shown by the light gray horizontal bars in Figure 2.5(b). This requires buffer memory for an additional $(k-1)(n-k)$ pixels but increases the computation to communication ratio to $\sim(2k^2 - 1)/2$. If this amount of workspace memory is unavailable, then the image can be processed in stripes with relatively small overhead.

For locations within $k/2$ pixels from the edge of the image, there is no source data "underneath" part of the kernel. Many heuristics exist to handle these edge cases, but they increase the complexity and decrease the regularity of the control flow. The weakness of the control flow support in accelerators mean that implementing such a heuristic could negatively impact the performance of 2D convolution. Fortunately, in an HMP-based system, the accelerator can simply calculate the edge pixels incorrectly and let the sequential processor correct them after the accelerator is done. To illustrate the expected performance of this algorithm, consider a 7×7 kernel being applied to a 1280×720 HDTV image. Using the sequential processor to compute the edge pixels, roughly 1.3% of the image will need to be recomputed. For the remaining 98.7% of the image, by fully pipelining the implementation, the average parallelism is $97 = (2(7^2) - 1)$ operations, with a communication requirement of 2 pixels per cycle.

Rijndael is the cryptographic algorithm that was chosen for the advanced encryption standard (AES), and is part of the MiBench benchmark suite. Rijndael has three main

components, the cipher (encryption), the inverse cipher (decryption), and key expansion. The computational requirements of the inverse cipher are very similar to that of the cipher; the differences are mainly in the order in which operations are applied. Since the key expansion routine is run infrequently, when the key is changed, and it contains irregular data and control flow, it is relegated to the processor.

The algorithm as implemented in MiBench offers several implementation options that trade off memory and computation. The faster finite field arithmetic (FF_TABLE) option precomputes much of the core part of the computation and stores that information in a 2kilobyte table that can then be replicated to permit parallel access. The alternative, *i.e.* baseline, approach uses a small 256B table to perform the non-linear byte substitution but performs the rest of the computation as normal logical and arithmetic operations. The FF_TABLE implementation requires sixteen 2KB lookup tables and 48 operations per round while the baseline implementation requires sixteen 256B lookup tables and 136 operations per round. Thus, given sufficient workspace memory, the FF_TABLE approach offers better throughput and latency.

Given a modest size accelerator, it is reasonable to fully pipeline a single round for either implementation; this achieves an average parallelism of ∼9.6 operations per cycle and the core loop has a initiation interval of 14 cycles for baseline and 5 cycles for FF_TABLE. Ideally we could C-slow the entire encryption routine to overlap the computation of several text blocks, but the MiBench implementation uses cipher-block-chaining. This means that the cipher text from each stage is added to the subsequent plain text block, which prevents C-slowing.

2.4 Summary

The HMP model serves four roles:

1. It provides a common language for analyzing and comparing a variety of hybrid sequential/accelerator architectures.
2. Programmers can use the execution model to develop algorithms that run efficiently on accelerators that conform to the model without considering the details of any

particular architecture.

3. It defines a model for whole systems that include sequential and accelerator subsystems. Again, this kind of standard allows programmers to design their systems without studying the details of hardware system integration issues.
4. It serves as a target model for designers implementing highly accessible hybrid computers.

The need for a model like this is based on several assumptions. First, we cannot expect programmers who want to accelerate their application to spend the time needed to become expert accelerator programmers with low-abstraction languages. Second, for the foreseeable future, and for *most* interesting applications, compilers will not be able to perform the program transformations that will produce efficient acceleration from conventional sequential code. Finally, given an appropriate model, programmers can analyze the potential for acceleration in an application and design algorithms and programs that execute efficiently on implementations of that model.

In the next chapter we look at Macah, the language that we designed to be the C of the parallel coprocessor accelerator model. C exposes the von Neumann model to the programmer fairly directly, with features like unrestricted pointers to a single large pool of main memory. Macah exposes the HMP model in a similar way. For example, code is explicitly partitioned between sequential parts and accelerator parts, and access to main memory is restricted in accelerator parts. This close connection between language and model is very useful because it gives the programmer language-level control over the most important features of programs running on an accelerator.

Chapter 3
MACAH AND THE MOSAIC TOOLCHAIN

Macah is a C-like language designed as part of the Mosaic project. In this chapter I describe the language, the Mosaic system it is a part of, and the important parts of our compilation strategy.

3.1 Macah and the HMP model

Among the factors that influenced the design of Macah, one of the most important was the HMP model described in Chapter 2. We felt that the language features had to have a logical connection to the model so that Macah programmers could make good decisions about their programs without knowing the details of a particular compiler or architecture.

We introduce the features of the language and briefly discuss the connections to the HMP model in this section. Later we cover the language in greater detail in the context of a running example application.

- Kernel blocks look like regular blocks of C code, but they are marked with the new keyword `kernel`. The connection with the HMP model is simple: code inside kernels is accelerated, and code outside kernels runs sequentially. Some research projects for programming accelerators attempt to identify kernels automatically (for example, [GDKG05]), but the number of kernels in a typical application is small and identifying them is not challenging for programmers.

  There are some restrictions on what kind of code is allowed inside kernels; for example, calls through function pointers are not allowed. Compared to many C-like languages for accelerators, Macah allows a wide range of control flow patterns, thanks to the enhanced loop flattening transformation described in Chapter 4.

  The semantics of the language are relaxed somewhat inside kernels to allow for automatic loop pipelining. The details of this relaxation and its consequences are described

in Chapter 7.

- FOR loops are similar to for loops, except the compiler will automatically attempt to completely unroll FOR loops. FOR loops make writing a large number of parallel operations easier.
- Streams are first-in/first-out buffered data channels between threads or tasks. There are two distinct uses of streams in Macah. Almost all applications use streams to read from and write back to main memory. Kernel code is not allowed to read and write main memory directly, so we set up "memory accessor tasks" that perform the main memory reads and writes, and communicate with kernels via streams. This arrangement has two benefits: first, the fact that a completely different mechanism is used in kernel code to access main memory emphasizes the fact that such access is a precious resource. Second, we completely avoid having to perform alias analysis and detailed array analysis by having the programmer partition main memory access out of the kernel.

  The second use for streams is as communication channels between tasks that are both doing real computation, not just reading or writing memory. Here streams provide decoupling and buffering between tasks to avoid unnecessary synchronization.
- Tasks are essentially stylized threads. In our thinking about applications there are three different kinds of tasks, though the language does not currently syntactically distinguish between them. Kernel tasks run a kernel, perhaps after some initial sequential setup. Memory accessor tasks just read or write memory and communicate with another task over a stream. General sequential tasks do anything else that needs to be done, and are assumed to run on a conventional processor.
- Configurations are collections of streams and tasks. The first version of Macah and the Mosaic toolflow supported only configurations with a single kernel task, but we recently added support for multikernel applications. Configuration structure can be determined statically, which is important for accelerators like FPGAs where the layout of a configuration must be statically compiled.
- Tuning knobs are a special kind of constant that are defined to have not a single value, but a range of legal values. The system performs an empirical auto-tuning search for

good values for the tuning knobs in an application. Tuning knobs are critical for fitting an application to an architecture because the model does not reveal specifically how many processing elements are available or the size of the workspace memory. Our tuning knob search procedure is described in Chapter 6.

- Shiftable arrays are like C arrays, but they also support shift operators for moving the data up or down in the array. In addition to supporting many application domains nicely, shiftable arrays are useful for describing simple regular communication patterns between operations. Such communication patterns are good, because though the model does not specify a particular network style, communicating with a near neighbor is cheaper than communicating with a more distant processing element in most accelerators.

Standard C programs can be compiled and run as Macah programs, but the compiler will not automatically accelerate them. In order to compile Macah code to an accelerator, the programmer must explicitly define a *configuration*, which is a collection of asynchronously running *tasks* that communicate with each other over buffered *streams*. Some of the tasks will execute *kernels*, which are the core compute-intensive parts of the program that actually run on an accelerator.

The "hello world" of Macah is shown in Figure 3.1. The configuration block, which defines a set of asynchronous tasks and streams, begins on line 3. It is important for the compiler to know the communication structure of these cooperating tasks, and the configuration block is how the programmer declares this structure. All streams and tasks declared in a configuration block are assumed to be part of the same configuration.

Notice that part of the task declaration is a list of the streams that a task will interact with. Streams in Macah are point-to-point (single sender task, single receiver task), and the task/stream connections are statically specified as part of a configuration.

The kernel—which begins on line 13—is the part of the application that will be accelerated. In this simple example, there isn't much acceleration to do, but we will see more realistic examples soon.

The simplest real Macah applications have a single kernel that runs in a task and a couple

```
1   int main(int argc, char *argv)
2   {
3     configuration {
4       int stream s1 = stream_create(int, 10),
5                  s2 = stream_create(int, 10);
6       task t1 (output s1) {
7         for (int i = 0; i < 100; i++) {
8           s1 <! i;
9         }
10      } // task t1
11
12      task t2 (input s1, output s2) {
13        kernel {
14          int t;
15          for (int i = 0; i < 100; i++) {
16            t <? s1;
17            s2 <! t + 1;
18          }
19        } // kernel
20      } // task t2
21
22      task t3 (input s2) {
23        int t;
24        for (int i = 0; i < 100; i++) {
25          t <? s2;
26          printf("Hello world %i\n", t);
27        }
28      } // task t3
29    } // configuration
30  } // main
```

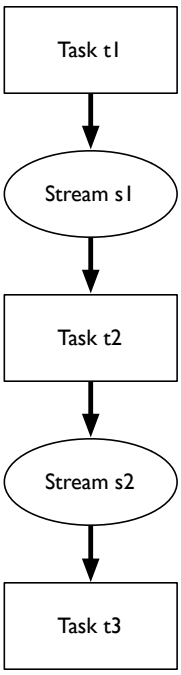


Figure 3.1: A very simple Macah program, along with the task and stream graph that is created when the configuration block is executed. Streams have exactly one sender task and one receiver task, but we generally represent them as nodes, not edges, in the task and stream graph because tasks can receive from and send to multiple streams and it is sometimes convenient to label the edges between tasks and streams with "port" information. `<!` and `<?` are the receive from stream and send to stream operators, covered in more detail later in the chapter.

of other tasks that just perform reads from or writes to main memory, and communicate with the kernel task over streams. These tasks and the streams they communicate over make up a single configuration.

Compared to many streaming languages, Macah gives the programmer a lot of control over the inter-task communication patterns in their program. Tasks can choose to execute sends and receives based on some dynamically computed condition, which means that Macah programs do not have to fit in the synchronous dataflow mold. Macah also has non-blocking sends and receives, which can succeed or fail depending on the state of the stream buffer at

runtime.

Macah is "C-level" in the sense that the balance between transparency and abstraction relative to machines that implement the hybrid accelerator model described in Chapter 2 is similar to the balance that C has relative to conventional sequential processors. There are certainly situations in which Macah programmers will have to look under the hood of the model to take advantage of specific features of particular architectures. However, they should be able to do most of their work with architecture-independent thinking.

So far we have only implemented Macah for the range of architectures the Mosaic system can simulate, which is a narrower set than all accelerators. My hope is that the language itself is flexible enough that it would not take a huge amount of effort to compile it to different families of accelerators. When writing Macah code, programmers do make choices about how to organize loops and buffer data that might work better on one style of accelerator than another. However, making these kinds of adjustments to port an application should be far less work than porting from an HDL to CUDA.

3.2 Example application: motion estimation

Much of the remainder of this chapter is organized around a running example, block matching motion estimation, which is the most computationally significant part of video compression for modern, high compression-ratio codecs. We use this application to describe in detail the novel features of Macah and our prototype compiler.

Block-matching motion estimation (BMME) is the part of video compression that finds similar blocks of pixels in different frames. Video codecs that support high compression ratios, like H.264, allow blocks of pixels in one frame (the *current* frame) to be defined as similar to some other block of pixels in another frame (the *reference* frame). The difference in position of the two blocks within their respective frames is called the *motion vector* (MV). The MV plus the small pixel value differences between the two blocks can be encoded in far fewer bits than the raw pixel data. Motion estimation terminology is illustrated in Figure 3.2.

During the compression process, the encoder must decide which block in the reference frame to use for each block in the current frame. This decision is made by the motion

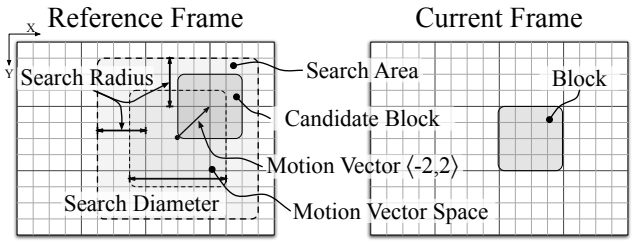


Figure 3.2: An illustration of some motion estimation terminology. In this picture, the frames are 16px×12px, the blocks are 4px×4px, and the search radius is 3px. The goal in motion estimation is to find the block of pixels inside the largest shaded region in the reference frame that most closely matches the shaded block of pixels in the current frame.

estimation algorithm. "Full search" (FS) is the simplest BMME algorithm. For each block in the current frame, it does a complete block comparison with every block in the reference frame that is within the *search radius* defined by the codec. This algorithm clearly finds the best MV in the window, but at a huge computational cost. For example, one 1920×1080 frame of full search with a search radius of 15 requires almost 2 *billion* pixel comparisons.

Fortunately, BMME can be approximated very accurately with heuristics that drastically reduce the amount of computation required. The variety of BMME heuristics that have been proposed is impressive, but most are based on a combination of four ideas. 1) Motion estimation can be performed on down-sampled versions of the input frames, with detailed block comparisons only done in regions that the down-sampled comparison judged to be promising. 2) Block comparisons for a sparse subset of MVs can be tested first, with more detailed searching in the area of the best comparisons. 3) "Predictive" BMME algorithms first try MVs based on which MVs were best for adjacent blocks (in both space and time), which works because of the strong spatial and temporal correlation of motion in most video. 4) Finally, the search for a good MV for a particular block can be terminated early as soon as a "good enough" MV is found. When carefully combined, these heuristics can reduce the computational demands of BMME by two to three *orders of magnitude* compared to FS,[1]

```
void motionEst(refFrame, curFrame, bestMVs)
{
  // outer two loops iterate over all blocks
  for (i=0; i<ImgH/BlkH; i++) {
    for (j=0; j<ImgW/BlkW; j++) {
      // initialization of the block difference array
      int blkDiffs[SrchDia][SrchDia];
      for (y=0; y<SrchDia; y++) {
        for (x=0; x<SrchDia; x++) {
          blkDiffs[y][x] = NOT_COMPUTED;
      } }
      boolean searching = 1;
      motion_vec_t mv;
      int bestD = infinity;
      while (searching) {
        chooseMV(blkDiffs, bestMVs, &mv);
        d = compareBlks(ref,cur,i,j,mv);
        if (d < bestD) {
          bestD = d;
          bestMVs[i][j] = mv;
        }
        blkDiffs[mv.i][mv.j] = d;
        searching = stillSearch(blkDiffs);
} } } }
```

Figure 3.3: Sketch of generic heuristic motion estimation in C. Particular heuristics are defined by the implementation of `chooseMV` and `stillSearch`. `chooseMV` decides which motion vector to test next. `stillSearch` decides when to stop searching for a particular block.

with negligible reduction in video quality[ZLCP04].

Heuristic approaches to BMME are extremely fast, but also relatively complex (a high-level sketch is shown in Figure 3.3). As a result, many researchers continue to use FS as a benchmark to demonstrate the power of coprocessor accelerators. But there is no reason to run FS on an accelerator when smarter algorithms can compute (almost) the same result at least as quickly on a conventional processor. Similar patterns exist in other application domains. For example, the BLAST tool uses a heuristic approach to compute the same biological sequence alignments as the Smith-Waterman algorithm in a fraction of the time, with only a small loss of accuracy. Just like the motion estimation example, BLAST is less predictable and regular than Smith-Waterman. Programming tools for accelerators must be able to handle these fast algorithms for the architectures to be relevant to the given application.

---

[1]The speedup factor depends strongly on the search radius.

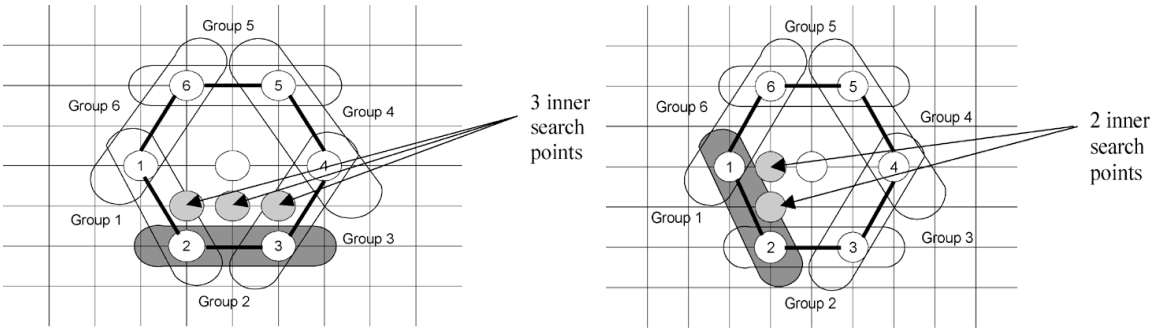


Figure 3.4: An illustration of the inner search points chosen by the enhanced hexagonal search algorithm. It is interesting to note the the number of inner points tested depends on which pair of outer points is best. Graphics borrowed from [ZLCP04].

3.2.1 Accelerating motion estimation

The particular BMME implementation that I chose to accelerate is called the enhanced hexagonal search (EHS)[ZLCP04]; other heuristic searches would have worked as well. EHS is a three phase algorithm. In the "predictive" phase, block comparisons are done for the $\langle 0, 0 \rangle$ MV and a small number of other MVs that were found to work well in previously processed adjacent blocks. The best of these MVs is taken as the initial center of the "coarse search" phase. In the coarse phase, block comparisons are done for the six MVs arranged in a hexagon around the current center MV. If any of them is better than the center, the best is taken as the new center and the process repeats. When the center is better than all of the points of the hexagon around it, the algorithm moves on to the "fine search" phase. In the fine phase (Figure 3.4), a few more blocks inside the perimeter of the final hexagon are compared. The best MV from the fine search phase is taken as the best MV for the block.

Three things are important about heuristic algorithms for motion estimation: 1) Even though they are much more efficient than FS, they still do a large amount of computation in the block comparisons. 2) Their control flow and data access patterns are highly dependent on the input data. 3) Intermediate results produced by the algorithm are used relatively quickly to make decisions about what to compute next. There is additional complexity in real video compression systems that we do not discuss in this chapter, including

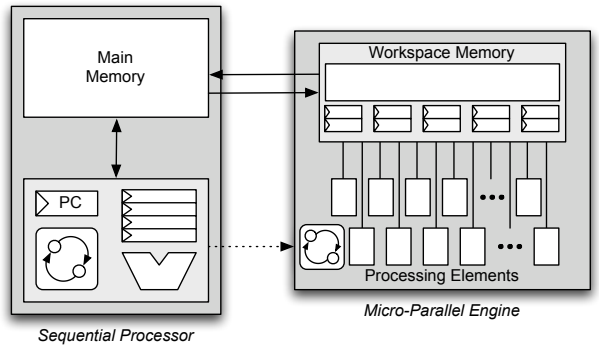


Figure 3.5: An abstract model of a hybrid processor-coprocessor system. Macah programmers need to think about writing accelerable code at the level of detail shown in this model.

variable block sizes, sub-pixel MVs, and multiple reference frames. This added complexity only strengthens the argument for support of sophisticated algorithms in accelerator programming languages and compilers.

Before implementing EHS in Macah, we will analyze its potential for acceleration. This analysis is done relative to the abstract model described in the previous chapter, an illustration of which is repeated in Figure 3.5. In order to write good Macah code, the programmer must do this kind of analysis, and therefore must have a high-level understanding of the structure and behavior of coprocessor accelerators. Though Macah looks like C, a well written version of EHS for a sequential processor will not produce efficient accelerator code.

Some of the most important constraints in the accelerator are the limited local memory and limited external communication bandwidth. The processing elements draw most of their input data from this local memory, because the bandwidth of the connection to the larger main memory is substantially lower than the computational throughput of the accelerator. Applications must have a sufficiently high computation to main memory bandwidth ratio in order to accelerate well.

EHS performs approximately 10 block comparisons per block on average (with a search radius of 7). Each block comparison requires $16 \times 16 = 256$ pixel comparisons. Each pixel comparison requires approximately 4 operations (2 reads, one absolute difference, and one

accumulation). So EHS requires about $10 \times 256 \times 4 = 10240$ operations per block.

The main memory bandwidth requirements depend on what is stored in workspace memory. On average, we need to transfer at least a block's worth of pixels for the current frame and the reference frame ($2 \times 16 \times 16 = 512$ pixels). Depending on how large and flexible the workspace memory is, we may have to transfer pixels from the reference frame multiple times. Optimistically assuming that each pixel is transfered only once, and assuming two bytes per pixel, that makes the main memory bandwidth requirement 1024 bytes per block. The computation to main memory bandwidth ratio comes to approximately 10 operations per byte. This number is reasonable, but leaves very little room for wasting memory bandwidth.

The next important feature of EHS that we consider is its complex control. Accelerator architectures have very poor support for unpredictable control flow, such as the logic to determine which block comparison to perform next. We believe that the best way to implement algorithms of this complexity is to partition them into a control part that executes on a conventional sequential processor and a kernel part that performs the repetitive, predictable piece of the computation. The control part sends commands consisting of block locations and motion vectors to the kernel part. The kernel part then does block comparisons and sends back computed block differences.

For such an implementation to work well, the accelerator must be integrated with a sequential processor. This could be die-level integration, as in FPGAs with embedded processors, or board-level, such as products from XtremeData, Inc. and DRC Computer Corp.

Next we must consider what data can be buffered in the workspace memory. A single frame of 1920×1080 video is almost 4MB of data (assuming 16 bits per pixel), and motion estimation requires data from at least two frames.[2] Real accelerators have a workspace memory capacity in the range of low hundreds of KBs to very low MBs, so realistically we will be able to store only a modest number of blocks worth of data in the workspace memory at a given time. This will affect how we do buffering in the Macah code.

---

[2]More advanced motion estimation algorithms perform comparisons with multiple frames simultaneously.

Accelerators work by executing many simple operations concurrently on their simple processing elements (PEs), so we have to think about which operations can execute in parallel. The inner loops that perform a single block comparison are a simple reduction, so they will parallelize nicely. The only complication is the order in which pixels from the reference frame are accessed depends on the MV currently being tested. This fact will make the buffer for the reference frame slightly more complicated than the buffer for the current frame.

Finally, there is inevitably some latency involved in sending an MV from the sequential processor to the coprocessor and getting a result back. Therefore, we want to have multiple MVs "in flight" at any time to keep the whole pipeline full. However, at certain points in the motion estimation algorithm, there may be only one or two new MVs to perform block comparisons on before those results are needed to decide what to do next. To keep the pipeline full, we need to work on multiple blocks from the current frame simultaneously.

This requirement forces us to change the algorithm because the predictive part of the sequential version needs to know what the best MVs are for its neighboring blocks. This change illustrates one of the most important weaknesses of a pure optimizing compiler approach to programming accelerators. Even if we assume that a compiler's loop and array analyses are smart enough to optimize irregular, input dependent code well, we generally expect compilers not to change the meaning of a program. We believe that this kind of super-aggressive optimization is best done in a separate code restructuring tool.

The accelerated implementation of EHS that we have sketched here is more complicated than the sequential version. The Macah code, parts of which are presented in the next section, is longer and more complicated than the C version as well. However, current compiler technology cannot automatically transform the plain C version into the most accelerator-optimized version. Also, it is not clear how to program this kind of fast motion estimation algorithm in the more abstract languages discussed in Section 1.4. Our only remaining option is to use an accelerator's native language, which forces the programmer to think at an even more detailed level about the hardware in the coprocessor. Accelerator research continues to include simple algorithms like full search, because more efficient versions require much more effort. There is some work on fast motion estimation on GPUs (for example,

```
#TuningKnob NumBlks int (1,16)

void motionEstAccelerated(refFrame,
                          curFrame, bestMVs)
{
  configuration {
    cmd_t stream cStrm =
      stream_create(cmd_t,10);
    rslt_t stream rStrm =
      stream_create(rslt_t,10);
    blkCompareSetup(refFrame, curFrame,
                    cStrm, rStrm);

    task mvChooseTask (input rStrm,
                       output cStrm) {
      mvChooseFn(refFrame, curFrame,
                 cStrm, rStrm, bestMVs);
} } }
```

```
void blkCompareSetup(refFrame,curFrame,cStrm,rStrm)
{
  pixel stream refStrm = stream_create(pixel,100);
  pixel stream curStrm = stream_create(pixel,100);

  task refFrameTask (output refStrm) {
    refReader(refStrm, refFrame);
  }

  task curFrameTask (output curStrm) {
    curReader(curStrm, curFrame);
  }

  task blkCompareTask (output rStrm, input cStrm,
      input refStrm, input curStrm) {
    compareBlocks(rStrm, cStrm, refStrm, curStrm);
  }
}
```

Figure 3.6: Top level of an accelerated Macah implementation of heuristic motion estimation. The application has been factored into two tasks: one for just comparing blocks of pixels and one for deciding which blocks should be compared. The two tasks communicate to each other over the streams `cStrm` and `rStrm`. One of the task instantiations occurs directly in this block, while the other is in the body of the function `blkCompareSetup`, which sets up the memory accessor tasks and streams for the block comparison kernel.

[TPO05]). However, all implementations only use resampling techniques, which are more data parallel and not the most efficient.

3.3 Motion estimation in Macah

The top level of our Macah implementation of motion estimation is shown in Figure 3.6. The main software architectural difference between the sequential version and the Macah version is that we have factored the algorithm into two tasks: one for comparing blocks of pixels, and one for choosing which motion vectors to compare. The two pieces of functionality will run in different tasks and communicate over streams.

3.3.1 Configurations

Configurations in Macah are essentially containers for tasks and streams, the details of both of which are covered directly below. Tasks are blocks of code that execute asynchronously

and communicate with each other over streams.

The standard Macah compilation flow has two distinct steps: first there is a conventional compilation, then there is a separate "configuration time" during which the configuration blocks are executed to determine the structure of the task and stream graph, but the tasks themselves are not executed. Because it is executed pre-runtime, the code inside of configuration blocks but outside of any particular task should be thought of as meta-programming code that declares tasks and streams, but does not perform application logic.

It is legal to write essentially arbitrary configuration code. For example, a configure block could open a file to read some parameters that are used to determine how many tasks and streams should be created. It is just important to remember that whatever it does, the configuration code is conceptually executed between conventional compile time and runtime. At runtime when the a program gets to a configure block the task and stream graph is executed. Buffer space is allocated for all the streams, and threads are spawned for all the tasks. Once all the tasks that participate in a configuration complete, execution resumes in the code directly after the configure block.

It is perfectly legal to call functions inside of configure blocks (Figure 3.6, line 9). The bodies of functions called at configure time can declare more streams and tasks.

3.3.2 Tasks

A task is a block of code that belongs to some configuration. When a configuration is actually executed, all of its tasks are launched simultaneously. Some tasks contain kernel blocks which run on the accelerator proper, some perform memory access and will ideally be compiled into direct memory access (DMA) commands, and others will run on a conventional sequential processor. During configuration time the compiler analyzes each of the tasks in a configuration and decides where it will execute on a particular architecture.

Because tasks each logically execute in a separate thread, there are some restrictions on how they are allowed to access data. Tasks are not allowed to directly access non-const local variables declared outside their own scope. The following is *illegal* Macah code:

```
1    int a = 42;
2    configure {
3      task taskName1 () {
4        a = 17;
5      }
6
7      task taskName2 () {
8        printf("%d n", a);
9      }
10   }
```

The reason for this is a language implementation issue that Macah inherits from C. `a` is allocated in the stack frame of the function that this code lives in, but the tasks run in separate threads with their own stacks. There is no direct way for the tasks to access `a`. Luckily there is a work-around. Tasks are allowed to share immutable (`const`) variables, because their value can be copied into the task's stack at task creation time, so `a` can be accessed with code like the following:

```
1    int a = 42;
2    int *const aPtr = &a;
3    configure_tasks {
4      task taskName1 () {
5        *aPtr = 17;
6      }
7
8      task taskName2 () {
9        printf("%d n", *aPtr);
10     }
11   }
```

In this case, the tasks are not modifying the (`const`) variable `aPtr` itself, so the code will work fine. There is still a race condition between the update through `aPtr` and the read through `aPtr`, but that gets into general shared memory multithreaded programming topics, which are well beyond the scope of this dissertation.

3.3.3 Streams

Streams are first-in, first-out channels between different threads or tasks that are used for sending data and certain kinds of synchronization. There are two main patterns for stream use in Macah programs. Simple Macah configurations that have a single main compute task with a kernel use streams to get data to and from main memory. In this case the streams are connected to very simple "memory accessor tasks". Ideally the memory accessor task and stream pattern can be recognized by the compiler and transformed into an optimized architecture-dependent memory read or write implementation of some kind. Our current

implementation runs all non-kernel tasks on a conventional sequential processor.

More complex Macah applications that have several concurrent compute tasks can use streams to directly communicate between tasks, instead of communicating through memory. The ordering of stream operations inside of kernels is handled in a somewhat relaxed way;[3] this is discussed in more detail in the kernel section below.

The basic operations that streams support are send (written *stream-exp* <! *exp*) and receive (written *l-exp* <? *stream-exp*). Both blocking and non-blocking versions of send and receive are available. The blocking versions will wait until the stream is not empty (for a receive) or not full (for a send) before returning. The non-blocking versions are ternary operators (receive: $l\text{-}exp_1$ :: $l\text{-}exp_2$ <? *stream-exp*) (send: *l-exp* :: *stream-exp* <! *exp*) that will attempt to do the stream operation, but will return immediately if it's not possible. The *l-exp* to the left of "::" will be set to true or false, depending on whether the operation completed successfully.

The type declaration of a stream carrying some type of data value (I'll use `float` in this example) should be written "`float stream s`" where `stream` is syntactically similar to the `*` in pointer type declarations. To work around a small parser problem, the current implementation of Macah actually uses the syntax "`float s` <∼∼<", which is more like the postfix array type declaration ("`float arr[10]`").

Streams are created with calls to intrinsic functions. The `stream_create` function (Figure 3.6, lines 7-10 on the left; lines 3 and 4 on the right), builds a stream capable of carrying data elements of the given type with the specified amount of buffering. All streams declared as part of a configuration are implicitly deallocated at the end of a configuration execution; if any data is left in a stream buffer at that time, it is lost.

The implementation of one of the memory accessor tasks for motion estimation is shown in Figure 3.7. As shown, functions can be used to separate the task declaration of a memory accessor task from its implementation, but this is not necessary.

Macah streams are unlike streams in languages like StreamIt [TKA02], and StreamC [KRD+03]. In those languages, by definition kernels consume and produce a particular

[3] "Relaxed" in the sense of relaxed memory models.

```
1   void refReader (pixel stream s,
2                   pixel refFrame[ImgH][ImgW]) {
3     int iPx, jPx, i, j;
4     for (iPx=0; iPx<ImgH; iPx+=BlkH*NumBlks) {
5       for (i = MAX(0, iPx-SrchRad);
6            i < MIN(ImgH, iPx+NumBlks*BlkH+SrchRad);
7            i++) {
8         for (j = 0; j < SrchRad; j++) {
9           s <! refFrame[i][j];
10      } }
11      for (jPx=SrchRad; jPx<ImgW; jPx+=BlkW) {
12        for (i = MAX(0, iPx-SrchRad);
13             i < MIN(ImgH, iPx+NumBlks*BlkH+SrchRad);
14             i++) {
15          for (j = jPx; j < MIN(ImgW,jPx+BlkW); j++) {
16            s <! refFrame[i][j];
17  } } } } }
```

```
void curReader (pixel stream s,
                pixel curFrame[ImgH][ImgW]) {
  int iBlk, jBlk, b, i, j;
  for (iBlk=0; iBlk<ImgH/BlkH; iBlk+=NumBlks) {
    for (jBlk=0; jBlk<ImgW/BlkW + NumBlks - 1; jBlk++) {
      for (b = MAX(0, jBlk + 1 - ImgH/BlkH);
           b < MIN(NumBlks, jBlk + 1);
           b++) {
        if (iBlk + b < ImgH/BlkH) {
          for (i=0; i<BlkH; i++) {
            for (j=0; j<BlkW; j++) {
              int i2 = (iBlk+b) * BlkH + i;
              int j2 = (jBlk-b) * BlkW + j;
              s <! curFrame[i2][j2];
} } } } } } }
```

Figure 3.7: Memory accessor functions for accelerated motion estimation. These functions are run in separate tasks; they feed the kernel through streams.

number of stream elements per *firing*. In other words, they have no send and receive operators that can execute conditionally. This more restrictive use of streams gives the compiler more opportunity to statically analyze the interactions of a group of kernels, but makes some programming styles difficult or impossible to use. For example, it is not clear how to program the motion estimation kernel that conditionally receives data into its buffers when it gets the command to move to the next block.

3.3.4 Tuning knobs

Macah is intended to be as portable as possible, but deciding how to structure a kernel to best exploit the local memory, external bandwidth and parallel computation resources of a particular accelerator often requires non-trivial application-level tradeoffs. Tuning knobs give programmers a tool to write code that can be automatically adapted to different architectures. They are typically used to control code features like the size of buffer arrays and the extent of loops. They are declared by the programmer (Figure 3.6, line 1). The compiler then searches for a value in this range that produces efficient code.

In the motion estimation code, the tuning knob `NumBlks` (for example, Figure 3.8, lines 3, 5, 8) controls the number of blocks from the current frame that are are buffered in workspace memory at a time. The size of the current frame buffer and reference frame buffer both depend directly on `NumBlks`. By using a tuning knob instead of a fixed constant

```
1   void mvChooseFn(refFrame,curFrame,cStrm,rStrm,bestMVs)
2   {
3     int blkDiffs[NumBlks][SrchDia][SrchDia];
4     for (i=0; i<ImgH/BlkH; i += NumBlks) {
5       for (j=0; j<(ImgW/BlkW) + NumBlks - 1; j++) {
6         // initialize all block differences
7         int bestDs[NumBlks];
8         for (b=0; b<NumBlks; b++) {
9           bestDs[b] = infinity;
10          for (y=0; y<SrchDia; y++) {
11            for (x=0; x<SrchDia; x++) {
12              blkDiffs[b][y][x] = NOT_COMPUTED;
13        } } }
14        boolean searching = 1;
15        motion_vec_t mv;
16        while (searching) {
17          blockNum = chooseMV2(blkDiffs, i, j, bestDs, bestMVs, &mv);
18          cmd.code = COMPARE_BLOCKS;
19          cmd.i = mv.i;
20          cmd.j = mv.j;
21          cmd.b = blockNum;
22          cmdStrm <! cmd;
23          blkDiffs[blockNum][mv.i][mv.j] = IN_PROG;
24          searching = stillSearch(blkDiffs);
25        }
26        cmd.code = DONE_WITH_BLOCKS;
27        cmdStrm <! cmd;
28  } } }
```

Figure 3.8: The sequential part of the accelerated implementation.

the compiler can adapt the program to accelerators with significantly different amounts of workspace memory. This explicit technique gives a level of portability and automation that exceeds the *ad hoc* technique of manually tuning C pre-processor `#define` values for each architecture. Tuning knobs are discussed in much greater depth in Chapters 5 and 6.

The code in Figure 3.8 is the main part that runs on the sequential processor; it is very similar to the C version sketched in Figure 3.3. There are three substantial changes in the motion estimation code itself. Where the C version calls a function to do a block comparison for a MV, the Macah version sends a command to the kernel and marks that MV in the distortion table as currently being worked on. In the function for choosing the next MV to try (Figure 3.9), if the heuristic needs to know the distortion for a particular MV, and finds that entry marked with a `IN_PROG`, it blocks until the coprocessor sends back a result. Finally, there is an additional loop to let the sequential side send MVs from several blocks at the same time.

```
1    int chooseMV2(int blkDiffs[NumBlks][SrchDia][SrchDia],
2        int i, int j, int bestDs[NumBlks],
3        motion_vec_t bestMVs[ImgH/BlkH][ImgW/BlkW],
4        motion_vec_t *mv) {
5      ...
6      // complex logic to choose b, x and y
7      ...
8      if (blkDiffs[b][y][x] == IN_PROG) {
9        blkDiffs[b][y][x] <? rsltStrm;
10       if (blkDiffs[b][y][x] < bestDs[b]) {
11         bestDs[b] = blkDiffs[b][y][x]
12         bestMVs[i+b][j-b] = {x,y};
13       }
14     }
15     ...
16     return b;
17   }
```

Figure 3.9: `chooseMV2` implements the heuristics of a particular motion estimation algorithm. Because the interface between the sequential logic and the kernel is asynchronous, this code might find a MV in the distortion table that has been sent to the kernel, but for which a result has not yet returned. In that case, the code does a blocking receive on the result stream.

3.3.5 Kernel blocks

Kernel blocks mark what code should be run on the accelerator, and ease the challenge of pipelining loops by relaxing the order of evaluation rules. Kernel blocks are written like standard C blocks, preceded by the new keyword `kernel`. The Macah compiler attempts to generate an accelerated implementation for the code in kernel blocks. If such an implementation is not possible for whatever reason, the compiler will report either an error or a warning, along with diagnostic information to help the programmer understand why the block cannot be mapped to the accelerator. Code outside of kernel blocks is translated to standard C, as discussed below.

The motion estimation kernel is shown in Figure 3.10. A large part of the code (most of lines 11-43) is devoted to managing the local frame buffers. In fact, the code presented here is slightly simplified for clarity of presentation; the real version has a bit of additional complexity to ensure that the buffer arrays can be accessed in parallel in the main block comparison loop. We have found that many of the more complex Macah applications have a lot of buffer management code. This suggests that it might be valuable to build a library

```
1   #define RefBuffH (NumBlks*BlkH+2*SrchRad)
2   #define RefBuffW (NumBlks*BlkW+2*SrchRad)
3
4   void compareBlocks(rStrm, cStrm, refStrm, curStrm)
5   {
6     px curBuff[NumBlks][BlkH][BlkW];
7     px refBuff[RefBuffH][^RefBuffW^];
8     kernel {
9       for (iPx = 0, iBlk = 0; iPx < ImgH;
10           iPx += NumBlks * BlkH, iBlk += NumBlks) {
11        // start getting strip of ref buffer for a new row
12        for (i2 = MAX(0,    iPx-SrchRad);
13             i2 < MIN(ImgH, iPx+NumBlks*BlkH+SrchRad);
14             i2++) {
15          for (j = 0; j < SrchRad; j++)
16            refBuff[i2-iPx+SrchRad][j + NumBlks*BlkW + SrchRad]
17              <? refStrm;
18        }
19        // done getting strip
20        for (jPx = 0, jBlk = 0;
21             jBlk < (ImgW/BlkW) + NumBlks - 1;
22             jPx += BlkW, jBlk++) {
23          // start shifting ref buffer for new blocks
24          for (i2 = MAX(0,    iPx-SrchRad);
25               i2 < MIN(ImgH, iPx+NumBlks*BlkH+SrchRad);
26               i2++) {
27            int refI = i2-i+SrchRad;
28            refBuff[refI] <<= BlkW;
29            for (j2 = 0; j2 < BlkW; j2++)
30              int refJ = j2 + (NumBlks-1)*BlkW + 2*SrchRad;
31              if (jPx + j2 < ImgW)
32                refBuff[refI][refJ] <? refStrm;
33          } } }
34          // get curr buffer data for new blocks
35          for (i2Blk = MAX(0, jBlk-(ImgW/BlkW) + 1);
36               i2Blk < MIN(NumBlks, jBlk + 1);
37               i2Blk++) {
38            if (iBlk + i2Blk < ImgH/BlkH)
39              for (i2 = 0; i2 < BlkW; i2++) {
40                for (j2 = 0; j2 < BlkH; j2++) {
41                  curBuff[i2Blk][i2][j2] <? curStrm;
42          } } } }
43          // done shifting buffers
44          boolean doneWithBlock = 0;
45          while (!doneWithBlock) {
46            boolean recvCmd = 0;
47            while (!recvCmd) {
48              recvCmd :: cmd <? cmdStrm;
49            }
50            if (cmd.code == CMD_COMPARE_BLOCKS) {
51              FOR (i2 = 0; i2 < BlkH; i2++) {
52                blkDiffTmps[i2] = 0;
53                refI = (i2 + cmd.i + cmd.b*BlkH);
54                for (j2 = 0; j2 < BlkW; j2++) {
55                  refJ = j2 + cmd.j + cmd.b*BlkW;
56                  refPx = refBuf[refI][refJ];
57                  curPx = curBuf[cmd.b][i2][j2]
58                  blkDiffTmps[i2] += ABS(curPx - refPx);
59              } }
60              blkDiff = 0;
61              FOR (i2 = 0; i2 < BlkH; i2++)
62                blkDiff += blkDiffTmps[i2];
63              rsltStrm <! blkDiff;
64            }
65            else { // command is 'done with blocks'
66              doneWithBlock = 1;
67  } } } } } }
```

Figure 3.10: Block comparison kernel in Macah.

of common buffer management functions.

In order to find enough parallel operations to keep the PEs busy, most kernels need to be pipelined. This means that "later" loop iterations are started before "earlier" iterations have completed. The order of execution rules inside kernel blocks have been subtly relaxed to accommodate this pipelining. Consider the simple example illustrated in Figure 3.11. There is a loop with three operations: a receive, some computation, and a send. In the sequential implementation the first receive happens before the first send, which happens before the second receive, and so on. In the pipelined trace, however, the second receive happens before the first send. If the rest of the program that this code interacts with is expecting to receive a value on s2 before sending another value on s1, the pipelined implementation will cause the program to deadlock.

The motion estimation kernel has exactly this structure. The kernel receives a command on the command stream, computes a block difference, and sends back a result. Both streams are connected to a sequential thread that receives results, chooses what MV to try next and sends back commands. This circular stream communication structure has the potential to create deadlock, which is why the receive on the command stream in the kernel is non-blocking. If the latency of the pipelined kernel and the communication between the processor and accelerator is long enough that the sequential thread cannot keep the kernel filled with commands, the non-blocking receive will fail, and "bubbles" will automatically be introduced into the pipeline.

The semantics of Macah explicitly allow stream sends and receives in kernel blocks to happen "late" and "early", respectively, from the perspective of an outside observer. This relaxation permits the compiler to perform loop optimizations like pipelining without analyzing the other code that interacts with the kernel through streams. This semantic issue is covered in greater detail in Chapter 7. Informally, cyclic communication patterns through streams in which at least one of the participating tasks is a kernel, can lead to deadlock. Tools for analyzing whole Macah programs for safety of stream communication patterns could clearly offer helpful error checking. In the spirit of Macah's "C-levelness", the default is to trust the programmer on this point.

Syntactically, any Macah code can be written inside kernels. However, kernel code needs

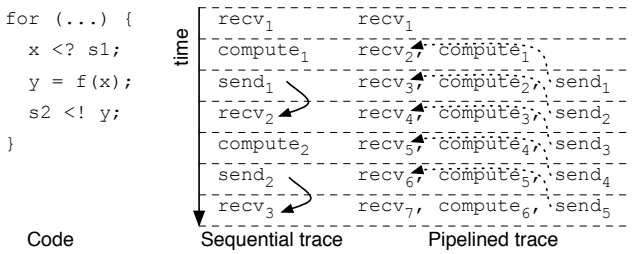


Figure 3.11: Simple pipelining example

to be "amenable to acceleration", which means that there are some restrictions on what what will work.

- All function calls must be in-lineable. In particular, the system does not currently support calls to recursive functions and calls through function pointers (though special cases of both could be supported). Most special system functions (or things that call system functions, like printf) cannot be called either.
- No "real" accesses through pointers are allowed. If `p` is some pointer, "`*p`" and "`p[4]`" are both disallowed.[4] In some cases it is convenient to take the address of some variable and then dereference the resulting pointer somewhere. For example:

```
1    int foo(float *f, ...)
2    {
3      S1;
4      *f = 4.5;
5      S2;
6    }
7    ...
8    {
9      float g;
10     kernel {
11       ...
12       int x = foo(&g, ...);
13     }
14   }
```

In this restricted case, using pointers is okay, because after inlining we get:

[4]Note that locally declared arrays are different from pointers (though C tries to make them look similar), and accesses to them are fine.

```
1     {
2       float g;
3       kernel {
4         ...
5         {
6           tempf = &g;
7           {
8             S1;
9             *tempf = 4.5;
10            S2;
11          }
12          x = retVal;
13        }
14      }
15    }
```

after constant propagation:

```
1     {
2       float g;
3       kernel {
4         ...
5         {
6           tempf = &g;
7           {
8             S1;
9             *(&g) = 4.5;
10            S2;
11          }
12          x = retVal;
13        }
14      }
15    }
```

after dereference-address-of simplification, and dead code elimination

```
1     {
2       float g;
3       kernel {
4         ...
5         {
6           {
7             S1;
8             g = 4.5;
9             S2;
10          }
11          x = retVal;
12        }
13      }
14    }
```

The pointer operations are all gone, which means that the kernel can be compiled without any problems.

- All data structures referred to inside a kernel will get allocated into the workspace memory. All such data structures must be statically allocated. If necessary, the

system will automatically copy data from main memory to workspace memory at the beginning of kernel execution, and copy it back at the end. This copying is performed when a variable (scalar or array) is accessed both inside and outside of kernel code.

- Global variable access is illegal.
- Non-local control flow, like exceptions and `return`, are not supported. With enhanced loop flattening (described in Chapter 4), it should be possible to support at least some kinds of non-local control flow. However, it has not been an important feature for the applications we have looked at.

The combined restrictions on data access in tasks and kernels mean that there is no way to directly share memory between kernels. Tasks can only share memory through pointer dereferencing, and pointer dereferencing is not allowed in kernels. Concurrently running kernels can only communicate with each other via streams. This restriction is intentional; it is not a consequence of immature compiler tools. Shared memory can be simulated by using a single task/kernel as the manager of the state, with other kernels sending requests and getting responses via streams. This is not a recommended pattern for most applications.

One common pattern in applications that requires some workaround because of these restrictions is getting an array of values into and/or out of a kernel. For example, many applications read an array of constant coefficients from a configuration file during startup and need to get these values into an array in a kernel. The standard workaround is shown. The standard workaround (Figure 3.12) is appropriate only for relatively small arrays that are intended to reside in workspace memory. Large arrays need to be streamed in and out as the kernel is running.

Finally, tasks in Macah are not explicitly declared by the programmer to be kernel tasks, memory accessor tasks or conventional processor tasks. A task is simply defined by what it does when it runs. This is different from languages like Impulse C, in which each task is declared as either “software” or “hardware”. The Macah approach is more convenient, but does require the compiler to do some inference.

```
1   int arr[N0][N1][N2];
2   int (* const arrPtr)[N0][N1][N2] = &arr;
3
4   configure {
5     task t1 () {
6       int arrBuf[N0][N1][N2];
7       // copy in
8       for (i0 = 0; i0 < N0; i0++) {
9         for (i1 = 0; i1 < N1; i1++) {
10          for (i2 = 0; i2 < N2; i2++) {
11            arrBuf[i0][i1][i2] = (*arrPtr)[i0][i1][i2];
12      } } }
13
14      kernel {
15        // ... use arrBuf freely ...
16      }
17
18      // copy out
19      for (i0 = 0; i0 < N0; i0++) {
20        for (i1 = 0; i1 < N1; i1++) {
21          for (i2 = 0; i2 < N2; i2++) {
22            (*arrPtr)[i0][i1][i2] = arrBuf[i0][i1][i2];
23      } } }
24    }
25  }
26  // ... arr will have the same values that arrBuf had at the end ...
```

Figure 3.12: The copy-in copy-out trick for working around data access restrictions in Macah.

3.3.6 Shiftable arrays

Shiftable arrays are just like C arrays, except they also support shift left and right operators (<<= and >>=, respectively). The result of shifting an array left by N is that each element of the array is moved to the left (that is, towards lower indices[5]) by N places. After a left shift of N, the right-most N places in the array are uninitialized. Right shifts work exactly the same way, in the opposite direction. Shifting by negative amounts is allowed; shifting in one direction by $-N$ is exactly the same as shifting in the opposite direction by N.

Shiftable arrays, in addition to being convenient for many application domains, help describe the kinds of regular local communication patterns that accelerators support well. The reference frame buffer in the motion estimation kernel is defined as a shiftable array

[5]It's a curiosity of computing conventions that the relationship between left/right and higher/lower indices is exactly the reverse for bits in a word.

because there is significant overlap between the search areas for adjacent blocks. When the kernel receives a command to move from one block to the next, it shifts the reference frame buffer by the block width and fills in the empty piece.

Shiftable arrays can be simulated with normal arrays and extra index arithmetic. However, shiftable arrays can be used to describe more directly the spatial relationships that exist in an algorithm, and potentially lead to a more efficient implementation.

In C, multidimensional arrays are just arrays of arrays. That is, an array of arrays is identical in almost all respects to an array of structs or floats. Shiftable arrays follow this C philosophy. Shiftable arrays of arrays (`arr[^N^][M]`) are legal, as are arrays of shiftable arrays (`arr[N][^M^]`) and shiftable arrays of shiftable arrays (`arr[^N^][^M^]`). If the "first dimension" is shiftable, then a statement like "`arr` <<= 3;" will move all the "second dimension" arrays by three slots. If the "second dimension" is shiftable, then a statement like "`arr`[`i`] <<= 3;" will move all the individual pieces of data in just the i-th "second dimensional" shiftable array. Rotation is similar to shifting, but the data that would have been shifted off the end and lost is instead copied into the slots that were vacated at the opposite end of the array. Rotation is not currently officially supported, but probably should be.

3.3.7 FOR loops

FOR loops are simply for loops that the user has declared should be unrolled completely at compile time. In the motion estimation kernel, the loops over the height of the block are all FOR loops (Figure 3.10, line 51). The iterations of a FOR loop need not be completely independent. The accumulations needed to produce a single distortion value for a block create a moderately long dependence chain, which is exactly why we rely on pipelining to overlap the computations of adjacent iterations.

Loop transformations like unrolling, interchange and blocking are well understood and can be performed by compilers. Advanced parallelizing and vectorizing compilers [AJ88] use sophisticated linear algebra-based loop and array analyses to decide how to apply loop optimizations. However, the extent to which they are applied can have a significant impact

on important issues like workspace memory and main memory bandwidth requirements, and it is far from trivial to decide automatically what combination of transformations will produce good results. The authors of [CDG+06], who include pioneers of parallelizing and vectorizing compilers, state "the quality of compiler-optimized code for high-performance applications is far behind what optimization and domain experts can achieve by hand... the performance gap has been widening over time". We therefore consider it important to give the programmer the tools needed to express which operations should be carried out in parallel. In the future, it may be worthwhile to use these kinds of optimizations on the most inner loops of applications like motion estimation.

3.4 Implementing Macah: Mosaic toolchain overview

The initial goal of the Mosaic project was to investigate the factors that contribute to energy efficiency in the design of coarse-grained reconfigurable architectures (CGRAs). The project has grown to include not only architecture, but circuit design, power modeling, compiler algorithms, programming language design and application programming. The main engineering artifact that we have produced is a toolflow (Figure 3.13) for compiling programs written in Macah and simulating them on CGRAs.

The box labeled "Electric VLSI Arch. Generator" represents a flexible system for specifying architectures. The primitive components that we currently support include arithmetic and logical units (ALUs) for primitive operation evaluation, multiplexors for dynamically controlling the flow of data, registers for short-term data storage, memories for long-term data storage, and I/O ports for implementing stream operations.

In order to support applications that have more operations than there are physical function units, each architectural primitive has a small configuration memory. The array has a single "program counter" that is broadcast to all the configuration memories and controls which operation each primitive performs. There is currently no support for branching or dynamic execution of different instructions based on live data, though we are actively investigating such features. Operations that cause side-effects (like stream sends and receives, and memory writes) have predicate inputs that dynamically control whether the operation should perform its effect or not.

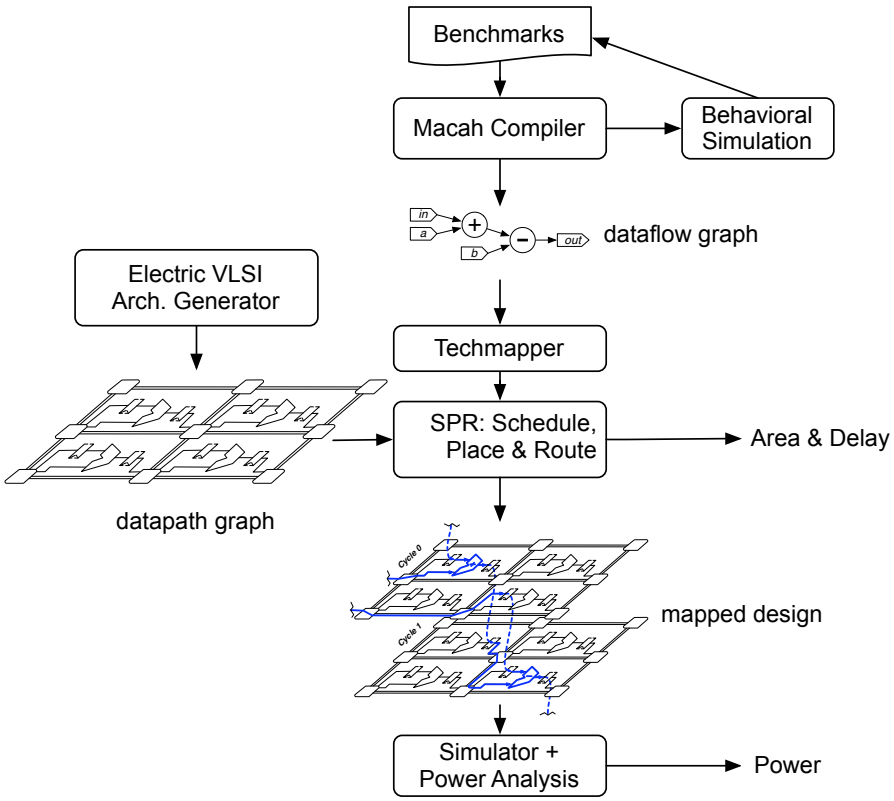


Figure 3.13: An overview of the Mosaic toolchain.

In the middle of the Mosaic toolchain diagram is SPR, which is responsible for mapping a dataflow graph representation of a kernel onto a particular architecture. SPR can be seen as a blend of a conventional software compiler back-end—scheduling and register allocation—and a conventional hardware CAD tool back-end—placement and routing. We will look more closely at how SPR works in section 3.6. The output of SPR is a configured architecture that can be simulated to estimate performance and energy consumption.

The top parts of the toolchain diagram—the front-end tools and benchmarks—are the main subject of this dissertation. The toolchain currently supports five different ways of running Macah programs, with different pros and cons.

Plain C. Macah programs can be translated into plain C with calls to a runtime library that provides implementations for the Macah features that do not exist in C. This

mode of execution provides no acceleration, but compilation is fast and it requires no additional infrastructure. It is primarily used for early-stage application development when the programmer is mostly focused on getting the logic of the application right.

Plain C with kernel instrumentation. The next step closer to a real implementation involves the compiler going through the architecture-independent optimization processes for the parts of the program that will eventually run on an accelerator. The program is then translated "back" into plain C with extra performance monitoring code in the kernels. Because it is still architecture-independent, this mode cannot provide a complete picture of how the program will perform on an accelerator, but some performance issues can be spotted early. Kernel instrumentation slows down the execution of the program somewhat relative to plain translation to C, but not by more than a factor of 2.

Architecture-independent Verilog. The interface between the front-end and SPR is dataflow graphs (DFGs) for the kernels. The DFGs are implemented as simulateable Verilog code, and the whole program can be run with the kernels executed in a Verilog simulator. The runtime library is responsible for starting the simulator and communicating the data back and forth.

In theory this style of simulation does not provide any more information than the "C with kernel instrumentation" mode, though there are some differences between the actual data gathering that has been implemented. The main benefit of having this mode of execution is in isolating bugs in the toolflow. When problems are identified we use the architecture-independent Verilog execution mode to determine quickly whether the problem is most likely in the application or front-end part of the compiler versus the architecture model or SPR. Verilog simulation is considerably slower than any of the modes that do not involve Verilog, at least by an order of magnitude. This mode is not frequently used for application development.

Architecture-dependent Verilog. The backend part of the compiler maps kernels onto the architecture it is given, and produces a configured version of the architecture in Verilog. In particular, all of the operations have been given slots in a schedule and registers and routing paths have been allocated to get data from one operation to another. The execution of the non-kernel code works just as with the unscheduled kernels. The only

difference from the sequential side's perspective is that the order in which send and receive requests come from the Verilog simulator may be different. This execution mode provides more implementation detail, but costs roughly another order of magnitude in execution time compared to architecture-independent Verilog.

Power modeling. The architecture-dependent simulation can be instrumented to measure signal activity in sufficient detail to provide reasonably accurate estimates of power consumption. This carries yet greater run time and memory costs, but is, of course, essential for experiments involving energy efficiency.

Notably absent from this list of implementation options is running kernels on an actual accelerator. While it would be interesting to do real-world performance comparisons of the Mosaic system versus other options, the specific research results in later chapters do not depend on that level of implementation detail. The performance results are given in terms of relative comparisons of different options within the Mosaic system.

Synthesizing the output of SPR to an FPGA should be mostly straightforward, and we believe the performance of such an implementation would be reasonable. The reason that we have not undertaken that project is that implementing the I/O and system integration components efficiently requires significant engineering effort.

3.5 Compiling Macah I: front-end

As indicated in the Mosaic toolchain system diagram (Figure 3.13), we have split compilation of Macah into two distinct pieces: an architecture-independent front-end and an architecture-dependent back-end (SPR). The front end is an extension of CIL, the C parsing and translation infrastructure [NMRW02]. We have modified the parser and internal data structures to accommodate Macah's new features. Most of the analyses and transformations described in this section are well known. What we provide here are explanations of why and how they are applied differently in the context of coprocessor accelerators. In CIL, all loops are represented as infinite loops with explicit breaks and continues. This works well for us, because we can implement the critical loop optimizations once for all kinds of loops.

3.5.1 Kernel partitioning

For each kernel block, the necessary control transfers between the sequential processor and the accelerator are automatically generated by the compiler, as are any data transfers that are necessary for data structures that are accessed both inside and outside of a kernel. This piece of compiler support is conceptually simple, but quite valuable, because systems that require kernel code and non-kernel code to be written in different languages create a large amount of manual interfacing work for the programmer.

3.5.2 Function inlining

Function inlining is a well known optimization, which replaces calls to a function with a copy of the body of the function. Complete function inlining is required for most kinds of accelerators, because they do not support function calls. Partial support for function calling on accelerators has been investigated, but there is significant tension between the dynamic allocation of stack frames that true function calling support implies and the limited nature of workspace memory.

3.5.3 FOR loop unrolling

A FOR loop is replaced by multiple copies of its body, with constants filled in for the loop induction variable. It is considered an error, if the initial value, termination condition or induction variable increment cannot be computed at compile time.

3.5.4 Array scalarization

Array scalarization breaks arrays up into smaller pieces that can be accessed independently, when it is legal to do so. The motion estimation code is carefully structured so that after FOR loop unrolling, both the current frame buffer and the reference frame buffer are accessed only by constants in their first dimension. It is then clear without any sophisticated array analyses that each sub-array can be allocated to a different physical memory and accessed in parallel. The `dists` array will be similarly scalarized.

```
if (e) {
  a = x*2;
  b = y/3;
  c <? s;
}
else {
  a = z+r;
  while (1) {
    ...
  }
}
```

(a) Before

```
eThen = e;
eElse = !eThen;
aT = x*2;
bT = y/3;
if (eThen)
  c <? s;
aE = z+r;
if (eElse)
  while (1) {
    ...
  }
a = eThen ? aT : aE;
b = eThen ? bT : b;
```

(b) After

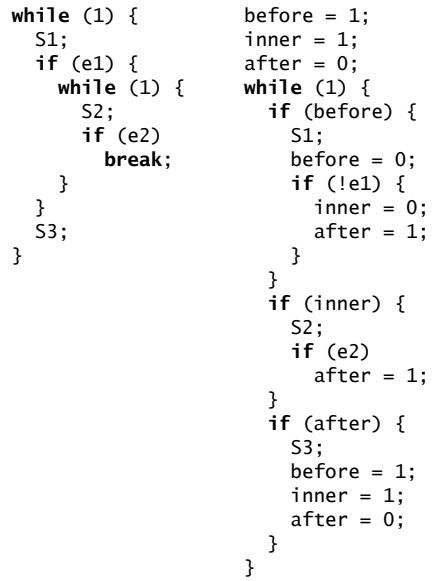
Figure 3.14: "If conversion" replaces conditional blocks with unconditional statements, selection expressions and individual predicated statements. This is almost always preferable for accelerators, which have poor support for unpredictable control flow.

Shiftable arrays that are not scalarized are implemented as normal arrays with additional offset and size variables. Indexing is performed relative to the offset, modulo the size, and shifting is implemented as offset arithmetic. If an architecture has built-in support for this kind of indexing, we take advantage of that.

What we call array scalarization is unrelated to another use of that term [ZK05b, ZK05a], which has to do with translating array expressions in data parallel languages into loops.

3.5.5 If-conversion

If-conversion is illustrated in Figure 3.14. After if-conversion, both sides of conditional branches are executed unconditionally. Variables that are modified on either side have to be renamed, with the final result selected after both sides have executed. Statements with side-effects, like the stream receive and the loop in the example, have to be individually predicated. Inside kernels, the current compiler completely converts all if-then-else and switch-case statements, though this can be an inefficient strategy. Aggressive if-conversion can lead to many values being computed and then discarded. Though it can be helpful to think about if-conversion as separate from any loop optimization, it is actually subsumed by loop flattening which is described briefly below, and in much greater depth in Chapter



```
while (1) {
  S1;
  if (e1) {
    while (1) {
      S2;
      if (e2)
        break;
    }
  }
  S3;
}
```

(a) Before

```
before = 1;
inner = 1;
after = 0;
while (1) {
  if (before) {
    S1;
    before = 0;
    if (!e1) {
      inner = 0;
      after = 1;
    }
  }
  if (inner) {
    S2;
    if (e2)
      after = 1;
  }
  if (after) {
    S3;
    before = 1;
    inner = 1;
    after = 0;
  }
}
```

(b) After

Figure 3.15: Loop flattening example for the special case where there is only one inner loop.

4.

3.5.6 Loop flattening

In order for kernels to perform well, the loops must be pipelined. The actual pipelining process is described below. Pipelining algorithms, like software pipelining [Lam88] and iterative modulo scheduling [Rau94a] can handle only a single loop, but Macah programs can have multiple nested and sequenced loops. We apply a transformation that in slightly different forms has been called flattening [GF95], coalescing [Pol87] and collapsing [KKLW80]. The basic idea is that the bodies of inner loops are placed directly into the outer loop and additional conditional guards are generated to implement the original control flow. In other work it is generally taken for granted that the loops involved have to be reasonably analyzable to avoid a large number of added conditional tests. However, we must flatten all loops in order to enable pipelining, so we generalized flattening to work with all kinds of loops. The intuition behind what flattening does is illustrated in Figures 3.15 and 3.16; the details

```
while (1) {
  S1;
  while (1) {
    S2;
    if (e1)
      break;
  }
  S3;
  while (1) {
    S4;
    if (e2)
      break;
  }
  S5;
}
```

(a) Before

```
count = 1;
while (1) {
  if (count == 1) {
    S1;
    count++;
  }
  if (count == 2) {
    S2;
    if (e1)
      count++;
  }
  if (count == 3) {
    S3;
    count++;
  }
  if (count == 4) {
    S4;
    if (e2)
      count++;
  }
  if (count == 5) {
    S5;
    count = 1;
  }
}
```

(b) After

Figure 3.16: Loop flattening example for multiple inner loops. These example loops are not predicated only to keep the example manageable. Our loop flattening algorithm can handle multiple predicated inner loops. Flattening with a single count variable, as shown here, is not a very efficient method. A more sophisticated method for flattening is covered in the next chapter.

are in Chapter 4.

3.5.7 Loop fusion

Loop fusion [KM94, Gan94, QCS02, LZSS04] (Figure 3.17) involves putting the bodies of multiple sequenced loops together into a single loop. Fusion can be very profitable for accelerators, because it takes code that would have run mostly sequentially and completely parallelizes it. If there are any dependencies between sequenced loops, fusion becomes much more complicated. In some cases it is possible to skew one or both loops by some number of iterations in order to make fusion possible. These transformations are not implemented in the prototype Macah compiler.

```
while (1) {
  S1;
  while (1) {
    S2;
    if (e1)
      break;
  }
  S3;
  while (1) {
    S4;
    if (e2)
      break;
  }
  S5;
}
```

(a) Before

```
while (1) {
  S1;
  brk1 = 0;
  brk2 = 0;
  while (1) {
    if (!brk1) {
      S2;
      if (e1)
        brk1 = 1;
    }
    if (!brk2) {
      S4;
      if (e2)
        brk2 = 1;
    }
    if (brk1 && brk2)
      break;
  }
  S3;
  S5;
}
```

(b) After

Figure 3.17: Loop fusion merges two (or more) sequenced loops into one. It can be applied only if there are no blocking dependences. In particular, S4 must not depend on S3 nor overwrite values that S3 depends on.

3.5.8 Setting tuning knobs

Searching for good tuning knob values is covered in depth in Chapter 6; here we make some comments about the integration of tuning knobs into the rest of the compiler flow. Tuning knob values can be used as true constants in Macah code; for example, they can be used to calculate the size of statically allocated arrays. Clearly tuning knob value choices have to happen before the rest of compilation, essentially during preprocessing.

3.5.9 Memory accessor streams

Memory accessor streams can be compiled into commands or "programs" for special memory interface units like direct memory access (DMA) controllers and streaming engines. This compilation process is not trivial, but because Macah programmers segregate the memory access code into memory accessor functions, it is at least clear what should be compiled this way.

(a)

```
configure {
  int stream s1, s2, s3;

  task t1 (input s1, input s2, output s3) {
    kernel {
      for (int i = 0; i < N; i++) {
        x = (? s1, A[i] ?);
        for (int j = 0; j < M; j++) {
          y = (? s2, B[i][j] ?);
          ...
        }
        z = (? s2, C[i] ?);
        ...
      }
      for (int i = 0; i < M; i++) {
        ...
        (! s3, D->arr[i] !) = e + f;
      }
    }
  }
}
```

(b)

```
configure {
  int stream s1, s2, s3;

  task t1 (input s1, input s2, output s3) {
    kernel {
      for (int i = 0; i < N; i++) {
        x <? s1;
        for (int j = 0; j < M; j++) {
          y <? s2;
          ...
        }
        z <? s2;
        ...
      }
      for (int i = 0; i < M; i++) {
        ...
        s3 <! = e + f;
      }
    }
  }

  task t2 (output s1) {
    for (int i = 0; i < N; i++) {
      s1 <! A[i];
    }
  }

  task t3 (output s2) {
    for (int i = 0; i < N; i++) {
      for (int j = 0; j < M; j++) {
        s2 <! B[i][j];
      }
      s2 <! C[i];
    }
  }

  task t4 (input s3) {
    for (int i = 0; i < M; i++) {
      D->arr[i] <? s3;
    }
  }
}
```

Figure 3.18: Streamable expressions are a convenience feature that let programmers avoid writing separate memory accessor tasks for simple memory access patterns. (a) is what the programmer writes; the streamable expressions (? ... ?) and (! ... !) are automatically replaced by the compiler with receives and sends as indicated in (b). The compiler also generates the memory accessor tasks using program slicing. Streamable expressions are currently only partially implemented in the prototype Macah compiler.

Streams and tasks provide a flexible mechanism for accessing memory in Macah. However, many applications have relatively simple memory access behaviors. For these applications, declaring streams and writing separate memory accessor tasks is fairly high code overhead. We have partially implemented a feature that we call "streamable expressions" in Macah that allows programmers to use normal array and pointer access code in their kernels. The expressions have to be specially marked, and the compiler then automatically generates the necessary streams and memory accessor tasks.

Figure 3.18 shows an example of the use of streamable expressions. Part (a) is what the programmer would write. A streamable read is written (? *stream-exp*, *exp* ?), and the meaning is "read *exp*, but do it in another task and stream the results here over stream *stream-exp*". Streamable writes are very similar, (! *stream-exp*, *l-exp* !), except the stream goes in the opposite direction, and *l-exp* has to be an expression that is legal for writing.

In order to generate the memory accessor tasks, the compiler uses program slicing [Wei81, HR92, BH93] on a copy of the original task to remove all statements and expressions that are not necessary for generating a particular streamable expression. Slicing is not computable in general, so there will be cases when the compiler has to report that extracting a particular streamable expression is not possible. In these cases the programmer will simply have to write the memory accessor task explicitly.

3.5.10 Inner functions

In Macah, functions can be declared inside other functions. Inner functions behave a little differently because the declared function actually becomes a closure, not a simple function. The biggest programmer-visible difference is that closures are not called with the usual syntax. In this example, `bar` is an inner function and `x` and `y` are local variables that the body of bar is allowed to access.

```
1     int foo()
2     {
3       int const x = 4;
4       float const y = 3.2;
5       ...
6       int bar(int w, float z)
7       {
8         return x * z + y * w;
9       }
10
11      int q = bar->f(bar, 3, 7.1);
12      ...
13    }
```

In the implementation, bar is actually turned into a pointer to a struct that has a field called "f" for the function itself, and (hidden) fields for the captured variables `x` and `y`. Calls to the function are actually done through the field f, and the closure itself has to be passed as the first argument so that the function has access to the captured variables. All captured variables must be declared const.

Inner functions are not used explicitly in much of the Macah code we have written. However, they are the mechanism we use to implement tasks. In implementation terms, tasks are translated into zero-argument inner functions that are then passed to thread creation functions when their configuration is run.

Inner functions in Macah are very similar to Apple's Blocks[CBl] extension to standard C. The nested function support in the GNU project's gcc is different, because it performs runtime code generation on the stack in order to support inner function calling without any syntax or type system changes[gcc].

3.6 Compiling Macah II: back-end

The back-end part of the Macah compiler is called SPR (schedule, place, route) [CE06, FCVE+09]. SPR is conceptually similar to conventional compiler back-ends that perform scheduling and register allocation. However, the physical location of an operation is important in accelerator architectures, so SPR has much in common with automatic circuit placement and routing tools.

In the back-end, architectures are represented as collections of interconnected ALUs, registers, local memories and stream ports. Communication delay within an architecture is non-uniform. The job of SPR is to schedule and place operations onto the appropriate

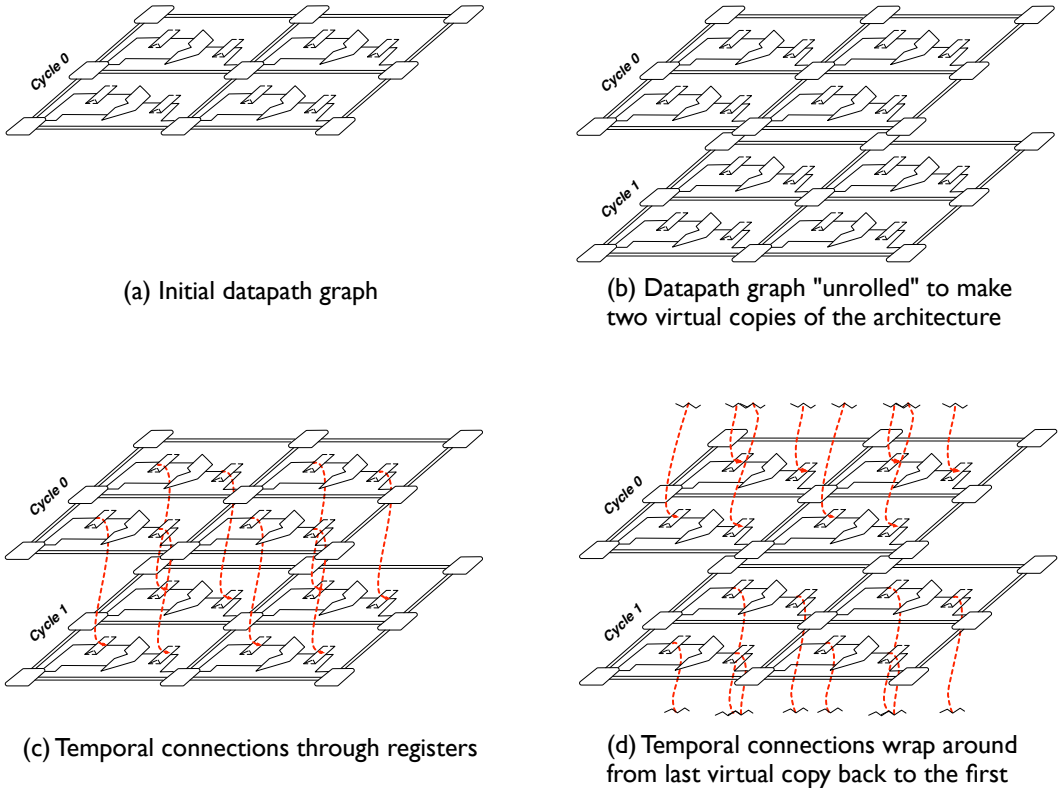
devices (arithmetic operations on ALUs, array reads and writes on memories, stream sends and receives on I/O ports), and configure the interconnect network to move data from producers to consumers. We expect that a wide range of accelerators can be faithfully modeled in this framework.

SPR accepts a single dataflow graph (DFG), which it treats as the body of an infinite loop with an explicit operation to indicate when to break. Ideally each operation would be placed on a hardware device and every cycle that device would perform that operation for the next iteration of the loop. However, there are two reasons that it might not be possible to start a new loop iteration every cycle. First, the DFG might not fit on the available hardware resources. Second, there might be inter-iteration dependencies in the code that force later iterations to wait until some part of an earlier iteration is complete.

To address these issues, SPR performs scheduling with software pipelining, which assigns every operation to a time step and decides how frequently new iterations of the loop will start. The time between iterations is referred to as the initiation interval (II). When the II is greater than 1 (which is common), each physical resource in the architecture will perform II different operations before it needs to start over again on its first operation for the next iteration. This time-multiplexing idea is illustrated in Figure 3.19, where II=2. Increasing the II makes more virtual hardware resources available and increases the amount of time between loop iterations, which makes it possible to respect inter-iteration dependencies.

The placement part of SPR uses a mostly conventional simulated annealing-based approach. However, the placement is on a version of the architecture that has been "unrolled" in time by a factor of II, as indicated in Figure 3.19(b). The schedule produced in the first stage of SPR determines which copy of the unrolled architecture an operation can be placed on. If an operation is scheduled in step S it must be placed on copy S mod II. The placer is allowed to adjust the schedule, but this introduces some additional complexity, since the placer needs to ensure that producers are scheduled before consumers.

The routing in SPR is actually a combination of conventional circuit routing and register allocation. If operation B depends on operation A, and A is scheduled at an earlier step in the schedule, the router has to use wires to get the data from the location of A to the location of B, and registers to delay the value for the appropriate amount of time. SPR



(a) Initial datapath graph

(b) Datapath graph "unrolled" to make two virtual copies of the architecture

(c) Temporal connections through registers

(d) Temporal connections wrap around from last virtual copy back to the first

Figure 3.19: SPR uses a software pipelining-style cyclic approach to scheduling. It makes a virtual copy of the architecture for each phase in the cyclic schedule. Connections between the different phases are made through registers. This arrangement allows scheduling/placement and register allocation/routing to be done simultaneously with mostly conventional CAD algorithms.

sees registers as connections between different instances of the architecture, as shown in Figure 3.19(c,d). With this unrolled architecture, SPR can use negotiated-congestion-based routing [ME95, LE04] to route and allocate registers simultaneously.

The back-end process outlined here has similarities to those used for other aggressively clustered VLIW processor and reconfigurable hardware projects, see, for example [FDF98, MVV+02, KFM06]. Other work has proposed solving the whole schedule/placement/routing problem as a single large optimization. We believe that breaking the problem up will help

SPR scale well to larger kernels and architectures.

3.7 Applications

In addition to motion estimation, we have worked on a variety of applications in Macah. As described in Chapter 1, applications for accelerators can be categorized as brute force, efficient or complex. Brute force applications have simple nested loops, lots of available parallelism and simple data access patterns. Efficient applications are still predictable but with more complex control flow and data access patterns, and often less parallelism. Complex applications use data-dependent branching and often more complex heuristics of some kind.

The applications for which members of the Mosaic group have done at least some work developing a Macah version appear in Table 3.1. Some of these applications are the classic bread and butter of accelerator language and architecture projects. Applications like FIR filter, 2D convolution, and matrix multiplication all have single kernels with simple nested loops and regular data access patterns. Implementing them is useful for performance comparisons, but these applications are simple enough that a wide range of languages and compilation strategies work reasonably well for them. Efficient applications like K-means clustering and principal components analysis (PCA) pushed us to find efficient implementations for applications with more complex control flow. Both of these applications have non-trivial nested and sequenced loops that are not amenable to conventional software pipelining.

Some of the applications motivated work on libraries that provide important functionality. For example, the pixel correlation application has a lot of data reuse and somewhat predictable access patterns. However, exactly which pixel is accessed at a particular point in the program is data-dependent, so the most basic streaming and buffering patterns that work for convolution and matrix multiplication are not applicable. What is needed to get the best performance out of this application is a banked cache, but many accelerators do not have hardware caches. Thus, we developed a parameterized software cache library that uses tuning knobs to adapt to the constraints of a particular application and architecture.

Our molecular dynamics implementation brought two language challenges to the fore.

Application	Category
Finite impulse response (FIR) filter	brute force
2D convolution	brute force
Discrete cosine transform for JPEG	brute force
Matched filter[BGT07]	brute force
Dense matrix-matrix multiplication	brute force
Image resolution scaling	brute force
Probabilistic Neural Network[CH99]	brute force
CORDIC[And98]	efficient
K-means clustering[GFM+03]	efficient
PET scanner event detection[HML+09]	efficient
Smith-Waterman sequence alignment[HJL+07]	efficient
Viterbi decoding[ZZH+09]	efficient
Wavelet encoding[FH05]	efficient
fast Fourier transform[HU05]	efficient
Pixel correlation	efficient
Principal components analysis[ZCS08]	efficient
Motion estimation for video encoding[RS01]	complex
Molecular dynamics simulation[SP06]	complex
BLAST sequence alignment[MUS05]	complex

Table 3.1: The applications for which we have written Macah code, ordered by increasing algorithmic complexity (a subjective judgment).

It was the first application we implemented that used feedback through streams, which has the potential to create unexpected deadlocks, as discussed in Chapter 7. It also has the kind of unbalanced paths—a fast common path and a slow uncommon case—that badly inflate initiation intervals during software pipelining. The loop transformation discussed in the next chapter was designed partially to address this issue.

Working with these applications has been an essential part of designing Macah and the Mosaic toolchain. Without a particular application to make a potential challenge concrete, it is hard to even know that it exists, let alone solve it.

3.8 Summary

We designed Macah to be a C-like programming language for accelerators with three primary constraints in mind:

- Macah reflects the HMP model defined in the previous chapter in the same way that C reflects the von Neumann model. This connection is valuable because it means that programmers can use the language effectively with a good understanding of the model, but without needing to understand the details of a particular architecture.
- The compilation strategy for Macah does not require any radical transformations. Designing the language such that reasonably simple compilation approaches work is important because it makes the connection between source code and the performance of a compiled program comprehensible to programmers. There are no parts of Macah compilation to accelerators that require the kind of pattern matching-based parallelization often employed by compilers from standard C to accelerators.
- Macah allows applications that *can* work well on accelerators to be coded without huge additional overhead. Members of the Mosaic group have worked on a reasonably wide range of applications, and though some adjustments have been made in response to programmer feedback, we have not found applications that do not work at all.

The loop flattening transformation described in the next chapter is important for freeing programmers to use whatever control flow patterns are natural for their application. Many C-like languages support only particular looping and branching control flow patterns, not

because it is impossible to support other patterns on accelerators, but because it is simpler for the compiler to recognize and handle restricted patterns.

The tuning features of Macah allow programmers to specify a range of legal values for "constants" in their programs. The compiler uses a search procedure, described in detail in Chapter 6, to find values that lead to good performance on particular architectures. This provides a degree of performance portability across accelerators.

Finally, in Chapter 7 we show that C-like languages that combine streaming I/O with compilers that use reordering optimizations like loop pipelining have serious semantic correctness issues. The semantics of Macah explicitly give the implementation flexibility to reorder stream operations on different streams.

Chapter 4

ENHANCED LOOP FLATTENING

Software pipelining[1] is a compiler transformation that statically schedules a loop such that operations from different iterations are overlapped in time. In other words, the first operation from a later iteration is scheduled before the last operation from an earlier iteration. The benefits of pipelining include avoiding stalls for long-latency operations like memory loads by increasing the distance between dependent operations, and exposing more opportunities for operation-level parallelism.

For compiling C-like languages to accelerators, pipelining is an essential optimization. Conventional approaches to pipelining work only on simple loops, which means that kernels that naturally have a more complex looping structure need to be manually recoded in order to get the full benefit of pipelining. This recoding process is quite tedious and error prone, even for kernels with only a few nested and/or sequenced loops. In this chapter I propose a new method for preprocessing loops for pipelining that can handle arbitrary static control flow and maintains the good performance characteristics of simple loop pipelining.

The earliest pipelining algorithms worked only on the simplest kinds of loops: a single loop with a constant number of iterations and no branching control flow. Supporting dynamic break conditions is a simple extension that is now in wide use. Acyclic control flow—for example if/then/else and switch/case—can be supported by applying if-conversion before pipelining. The idea behind if-conversion is that instead of executing one branch or another, both are executed unconditionally and the results are selected based on the branch condition. Other approaches to handling acyclic control flow are described in Section 4.1.

More complex cyclic control flow—for example, nested and sequenced loops—is a bigger challenge for pipelining than if/then/else control flow. One approach is to pipeline only the individual inner loops of a more complex loop nest. The problem with pipelining only inner

[1] In this chapter "software pipelining", "loop pipelining" and "pipelining" are used interchangeably.

loops is that pipelined loops have *prologue* and *epilogue* (or *fill* and *drain*) periods at the beginning and ending of the loop execution, and during those periods the parallel hardware of an accelerator is underutilized. Pipelining only the inner loops of a more complex loop nest means that each time execution flows from one inner loop to another, the epilogue of the previous loop generally cannot be overlapped with the prologue of the next loop.

There are some existing methods for pipelining nested loops such that prologues and epilogues can be overlapped, but the majority of them are applicable only to very restricted loop patterns. One approach that can handle arbitrary cyclic control flow is to first flatten the complex loop into a single-level loop and then apply pipelining to the flattened loop.[2]

Loop flattening is a transformation that "emulates" an arbitrarily complex loop nest with a single-level loop by guarding each part of the original loop nest with a different predicate. In each iteration of the flattened loop, only some of the predicates are true; additional logic is needed to make the flattened loop correctly implement the original nested looping structure.

Existing flattening algorithms are greedy in the sense that the predicate logic is set up to execute each operation in the earliest possible iteration of the flattened loop. This strategy minimizes the number of iterations of the flattened loop, which is clearly desirable, but it does not necessarily lead to the most efficient pipelined schedule. Inter-iteration dependencies are an important limit on the effectiveness of pipelining, and flattening a loop in a way that adds extra iterations can loosen the scheduling constraints imposed by inter-iteration dependencies and improve the overall pipelined schedule.

Enhanced loop flattening is a new framework we developed for flattening loops that makes it possible to control precisely the addition of extra iterations to create the most efficient balance of total number of loop iterations and good pipelined scheduling. Flattening a complex loop requires a fairly large number of predicate variables and Boolean operations to implement the predicate logic. Because of this, enhanced loop flattening is most appropriate for architectures that have hardware support for single-bit computation.

Section 4.1 covers background on conventional loop pipelining and existing methods

[2]Loop flattening is also sometimes referred to as loop collapsing or loop coalescing.

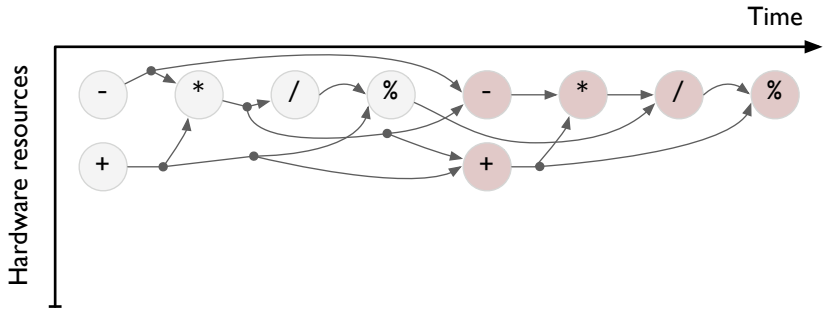


Figure 4.1: Without pipelining we cannot schedule operations from later iterations until all operations from earlier iterations have completed.

for extending pipelining to more complex loops. The high-level design of enhanced loop flattening is described in Section 4.2; the implementation details are given in Section 4.3. We evaluate enhanced loop flattening in the context of the Mosaic toolchain in Section 4.4. Some extensions and important issues related to enhanced loop flattening are discussed in Section 4.5.

4.1 Background

Software pipelining is a family of compiler methods for scheduling and execution unit allocation that convert loop-level parallelism into instruction-level parallelism. Pipelining exploits the fact that while the number of parallel operations available in a single iteration of a loop is often limited, operations from later iterations can be executed before earlier iterations have completed. Unlike complete loop parallelization, pipelining can handle loops that have inter-iteration dependencies. Here is a very simple example of a loop that can benefit from pipelining.

```
while (true)
  A := A + C
  B := B - C
  C := A * B
  D := C / E
  E := A % D
```

If we insist on completing all the operations from one iteration before any of the operations

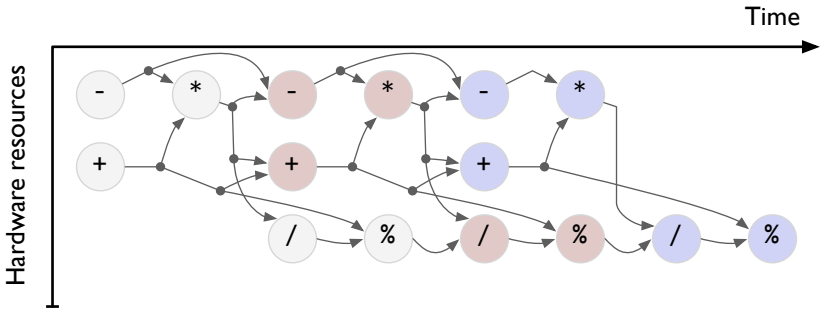


Figure 4.2: Loop pipelining allows later iterations to start before earlier iterations are complete.

from the next iteration can start, we cannot schedule the loop any tighter than the example schedule shown in Figure 4.1. There is a four-long chain of operations in a single iteration, which means that the latency of a single iteration of the loop cannot be any lower than four time units (assuming unit time latency for all operations).

Using software pipelining, we can get higher parallelism and throughput, as illustrated in Figure 4.2. Even though each iteration still takes 4 time units to complete, we can start a new iteration every 2 time units, roughly doubling overall throughput.

To help visualize pipelined loop schedules, "parallelogram diagrams", like Figures 4.3 and 4.5, are used throughout this chapter. In those diagrams a vertical slice is a snapshot of what each hardware unit is scheduled to do at a particular point in time; a horizontal slice is the complete schedule for a particular hardware unit; the schedule and hardware assignment of each individual iteration can be seen as a diagonal slice, though there is no actual requirement that the iterations line up on a diagonal. Figure 4.4 has a summary of the abbreviations we use.

The result of applying pipelining to a loop is a schedule that assigns every operation to a time step and hardware resource. It is assumed that every dynamic instance of a given static operation will execute at the same time relative to the start of its iteration, and on the same hardware resource. There are actually two distinct "schedules" in the context of pipelining. One is the schedule for a single iteration of the loop. In our simple example,

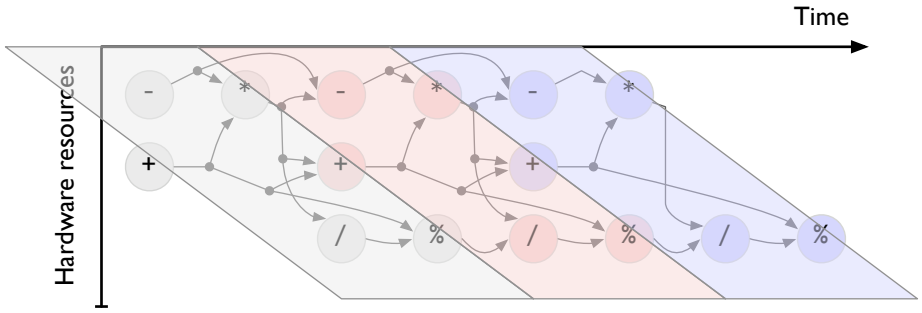


Figure 4.3: In "parallelogram diagrams" like this a vertical slice represents a snapshot in time, a horizontal slice represents the schedule for a single hardware resource, and a diagonal slice represents a single loop iteration.

that is:

Time step	Operations
0	$-$, $+$
1	$*$
2	$/$
3	%

The length of the single iteration schedule is also referred to as the *latency* of the loop. The other schedule that is important is the *steady state* schedule, which is the repeating pattern of operations executed by the machine at a particular time. In general operations from multiple iterations can overlap in the steady state schedule. In our simple example the steady state schedule is:

Time step	Ops from Iteration $N-1$	Ops from Iteration N
0	$/$	$-$, $+$
1	%	$*$

The length of the steady state schedule is the same as the amount of time between new iterations starting. This time is referred to as the *initiation interval* (II) of a pipelined schedule. Pipelined loops cannot start executing the full steady state schedule from the very first iteration, because executing operations for iterations before the first does not

Abbr.	Meaning
L	Latency (of a single loop iteration)
II	Initiation interval
T	Time
C	Trip count
RecII	Recurrence initiation interval
ResII	Resource initiation interval

Figure 4.4: Abbreviations used throughout the chapter.

make any sense. For this reason, pipelined loops have a *prologue* or *fill* period during which the first few iterations start executing, but the steady state schedule has not started yet. The length of the prologue is approximately equal to the latency of the loop. There is a symmetric period at the end of the execution of a pipelined loop called the *epilogue* or *drain* period.

The II is an important metric for pipelined loops, because the total execution time for all iterations of a pipelined loop (ignoring stalls) is the latency plus (II multiplied by one less than the total number of iterations). The total number of iterations is also called the *trip count*, and as long as it is reasonably high, the II is much more important that the latency for determining the performance of a pipelined loop.

There are two important limitations on the achievable II: resource limits and inter-iteration feedback dependencies. If a loop body requires N of some resource, but there are only M ($<N$) available in the architecture, the II must be at least N/M. This applies to each different kind of resource in the architecture, and the lower bound on the II imposed by resource constraints—referred to as the *minimum resource II*—is the maximum across all kinds of resources.

Perhaps less obviously, if some static operation depends (directly or indirectly) on an execution of itself from an earlier iteration, the latency of the dependency chain between those two executions also limits the II. In our simple example, the multiplication operation depends on the addition, which in turn depends on the multiplication from the previous

iteration. This means that the II must be long enough to execute an addition and a multiplication in sequence. In general, an inter-iteration feedback dependency chain can cross multiple iterations, and the bound imposed on the II—referred to as the *minimum recurrence II*—is equal to the latency of the chain divided by the number of iterations it crosses.

In the context of massively parallel architectures, computational resources are sufficiently abundant that for most applications we are more concerned about the minimum II imposed by feedback paths (the recII) than resource limits (the resII). Controlling the minimum recII is the main point of enhanced loop flattening. The details are covered later, but the intuition is that the iteration distance for specific inter-iteration dependency chains can be increased, which lowers the minimum recII imposed by that chain.

The *criticality* of a particular inter-iteration dependency chain is a measure of how close the minimum recII imposed by that chain is to the overall minimum II for the loop. The closer these two numbers are, the more critical the chain is. We can extend the notion of criticality from dependency chains to particular operations by defining the criticality of an operation to be the worst-case criticality over all inter-iteration feedback chains in which it participates.

The number of iterations that are "in-flight" at a particular point in time is an important measure of how deeply pipelined a loop is. This number is approximately equal to the latency of the loop divided by the II.

The basic terminology of pipelining is illustrated in Figure 4.5. The highlighted inter-iteration feedback path has a latency of 5 (assuming unit latency for all operations along the path) and an iteration distance of 2, which means the II can be no lower than $5/2 = 2.5$ time units. In most cases we measure time units in machine cycles, which means that fractional IIs are not possible with conventional pipelining algorithms.[3] For this example, the minimum integral II would be 3.

Finding enough parallel operations to make use of all the physical resources can be a serious challenge with parallel accelerators, and high IIs exacerbate this problem, because an application scheduled at II on an architecture with M physical resources needs $II \times M$

[3]Fractional IIs can be implemented, for example, by using a steady state schedule that executes multiple iterations.

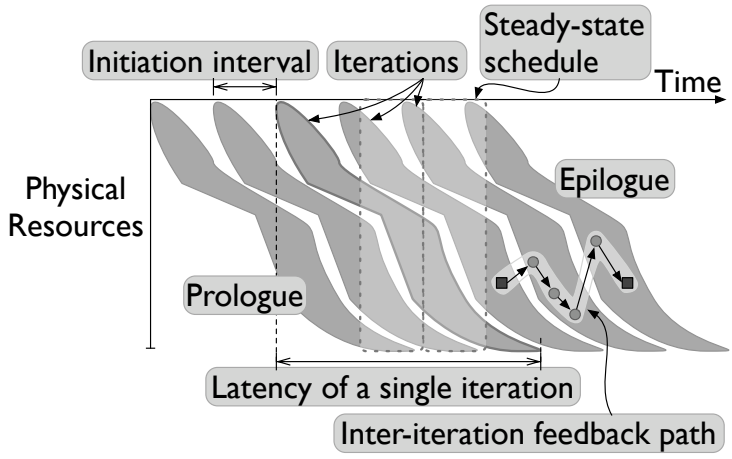


Figure 4.5: The basic terminology of loop pipelining. The repeated shape of the loop iterations signifies that in conventional software pipelining all iterations share the same schedule and resource assignments. The inter-iteration feedback path is the whole chain of operations from square to square. In this picture, 2 instances of the steady state schedule are shown, though typically there will be many more.

operations per iteration to achieve full utilization. To visualize why this is the case, imagine scheduling a loop on an architecture at some II as creating II virtual copies of the architecture to use.

For loops with very high trip counts, the latency of a single iteration is not usually important for overall program performance. However, the lower the trip count, the more important prologue and epilogue periods are. During prologue and epilogue, the hardware resources are underutilized, which negatively impacts performance.

If a pipelined loop is nested within another loop, as illustrated in Figure 4.6, and the prologue and epilogue periods of an inner loop can be overlapped properly, the total execution time will be approximately $C_{outer} \times C_{inner} \times II$. On the other hand, If the prologue and epilogue cannot be overlapped, the total execution time will be approximately $C_{outer} \times (L_{inner} + C_{inner} \times II)$.

Many different variants of pipelining exist; most require that the input to the pipelining

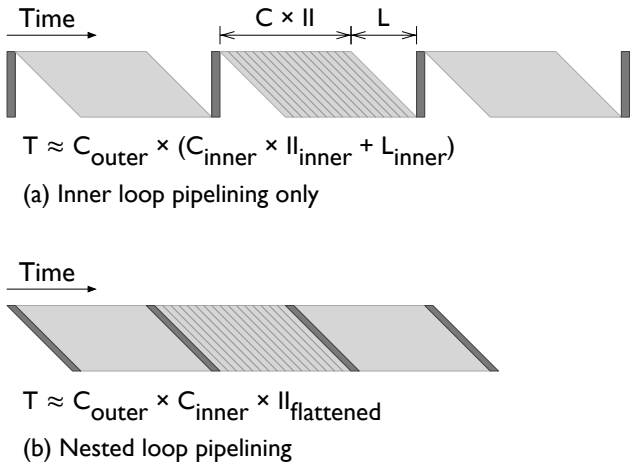


Figure 4.6: Sketches of the execution of a nested loop. In case (a), only the inner loop is pipelined, which results in the empty triangles at the beginning and end of each iteration of the outer loop. In case (b), the whole loop nest is pipelined together, which improves performance roughly in proportion to the ratio of $C_{inner} \times II$ to L.

algorithm be the body of a single loop with no function calls or non-trivial control flow. Iterative modulo scheduling [Rau94b] and swing modulo scheduling [LGAV96] are the classic implementations of inner-loop-only pipelining. No static pipelining algorithms we are aware of can handle truly dynamic control flow, like exceptions, continuations and calls through function pointers. Simple function calls are usually addressed with inlining before pipelining, and this is the assumption we make with enhanced loop flattening as well.

4.1.1 Previous work on pipelining complex loops

One existing approach to pipelining loops with complex control flow is *hierarchical reduction*[Lam88]. The hierarchical reduction (HR) method applies pipelining to more deeply nested blocks of code (like branches of an if/then/else or inner loop bodies), then treats the resulting schedule and resource allocation as a "complex primitive" and applies pipelining again to the next level.

The "reduction" in HR refers to reducible control flow, which is formally defined below.

Informally, programs that use structured control flow mechanisms will have only reducible control flow. Programs that use gotos or are optimized in certain ways can have *irreducible* control flow.

HR has two weaknesses compared to enhanced loop flattening:

- It is applicable only to reducible control flow.
- Good solutions to the inner/local scheduling and resource allocation problem can be quite suboptimal in the outer/global context.

Our compiler does not support the complex primitive concept required to implement HR, so we cannot experimentally evaluate the schedule quality that results from solving the scheduling problem hierarchically. However, the authors of [WMHR93] compared HR to predication in the context of if-conversion, and found that HR was significantly less efficient. Thus we expect that HR would produce poor results in our context as well.

Many other approaches to pipelining nested loops have been proposed [Ram94, YTZL97, BDH$^+$00, CW00, GSZ01, MD01, PHA02, RTG$^+$04, BRS07, RTG$^+$07, TCMC08, ZXQ$^+$08]. All these proposals are applicable only to limited classes of loop nests, like perfectly nested loops. There is some disagreement about the exact definition of perfect and imperfect loops. Loop nests with sequenced loops inside of other loops are imperfect by all definitions. Some authors consider loops nests where there is some non-looping code between outer and inner loops to be imperfect and while others consider them perfect.

Of the published approaches to statically pipelining complex loops, the one that covers the broadest class of loop nests is [FCT07]. Their approach can handle at least two levels of truly imperfectly nested loops. However, the prologues and epilogues of the inner loops cannot be overlapped if there are dependencies between the inner loops, which we will demonstrate is an unnecessarily strong restriction. Compared to this approach, our enhanced loop flattening can handle yet a broader class of loops (structured and unstructured, reducible and irreducible loops), and does a better job of overlapping epilogues and prologues in the presence of complex data dependencies.

Software Bubbles [GCHP02] is an interesting technique for software pipelining in the presence of hard to predict dependencies through memory. It has similarities with enhanced

loop flattening in that both methods add extra iterations to the execution of a loop in order to accommodate infrequent, long-latency events. However, the methods described in [GCHP02] are directly applicable only to a single loop, and it is not obvious how to extend them to address the issue of complex looping control flow.

A whole different perspective on pipelining complex loops is to delay much of the scheduling to runtime by augmenting the hardware with some form of dynamic token passing [BG02, Car05]. Dynamic approaches make it much easier to accommodate complex loops, but they also have non-trivial energy and run time overhead for the extra coordination that is required. Also, sharing of hardware between operations that execute under different conditions requires dynamic arbitration. This issue is discussed further below.

4.1.2 Flattening

Loop flattening transforms complex looping code into a single loop that can then be pipelined with conventional algorithms. Loop flattening has been proposed as a solution to a number of different problems [PW86, Pol87, vHK92, OD93, GF95, Kni98, KNP08].

Parts (a) and (b) of Figure 4.7 illustrate the effects of flattening. A complex loop is transformed into a single loop with additional predicate logic to control when the various blocks of code should execute. In each iteration of the flattened loop (part (b)), each block of code (S1, S2, S3) might execute, depending on the predicates (p1, p2, p3).

Each block can execute at most once in a single iteration of the flattened loop; flattening does not duplicate any blocks. Existing approaches to flattening have at least one of two limitations that we eliminate with enhanced loop flattening:

- Restrictions on the kinds of loops that are handled (such as perfectly nested loops).
- Being *greedy* in the sense that each operation is executed in the earliest possible iteration of the flattened loop. This may seem like a positive feature for performance, because the total number of iterations is minimized. However, in the context of loop pipelining it is not the best, because it can create long inter-iteration feedback paths.

Flattening and pipelining are complements to other loop optimizations, like unrolling or fusion, not a replacement for them. For example, if a program has two sequenced loops

(a)

```
do {
  do {
    S1;
  } while (C2);
  S2;
  do {
    S3;
  } while (C3);
} while (C1);
```

(b)

```
p1 := true, p2 := false,
p3 := false;
while (true)
    if (p1)
        S1;
        if (!C2)
            p1 := false;
            p2 := true;
    if (p2)
        S2;
        p2 := false;
        p3 := true;
    if (p3)
        S3;
        if (!C3)
            p3 := false;
            if (C1)
                p1 := true;
            else
                break;
```

(c)

```
p1 := true, p2_0 := false,
p2_1 := false, p2_2 := false,
p3 := false;
while (true)
    if (p1)
        S1;
        if (!C2)
            p1 := false;
            p2_0 := true;
    if (p2_2)
        p3 := true;
    p2_2 := p2_1;
    p2_1 := p2_0;
    if (p2_0)
        S2;
    p2_0 := false;
    if (p3)
        S3;
        if (!C3)
            p3 := false;
            if (C1)
                p1 := true;
            else
                break;
```

(d)

```
p1 := true, p2_0 := false,
p2_1 := false, p2_2 := false,
p3 := false;
while (true)
    (p1) S1;
    p1 := C2 ? p1 : false;
    p2_0 := C2 ? p2_0 : true;
    p2_2 := p2_1;
    p3 := p2_2 ? true : p3;
    p2_1 := p2_0;
    (p2_0) S2;
    p2_0 := false;
    (p3) S3;
    if (p3 && !C3 && !C1)
        break;
    p3 := p3 ? C3 : false;
    p1 := p3 ? !C3 : p1;
```

Figure 4.7: A simple loop nest (a) can be flattened in many different ways. (b) shows a greedy flattening, where each block executes in the earliest possible iteration of the flattened loop. (c) shows another possible flattening with extra predicate variables and extra "bubble" iterations between the two inner loops. These bubble iterations will help create slack in the schedule, will hopefully allow the loop to have a lower II, and only increase the trip count in proportion to the number of times the outer loop executes. (d) shows the completely flattened version of (c), which emphasizes the fact that all operations execute on every iteration of the flattened loop—some are just predicated off.

with similar trip counts and no true dependences between them, it is probably best to apply fusion first. Whatever other optimizations are applied, it is often still beneficial to use pipelining to perform the final scheduling.

Flattening control flow (with if-conversion or loop flattening) has the important side-effect that all operations are executed, whether their results are needed or not. The unnecessary executions can be a performance problem, which is discussed further in Section 4.5.

4.2 Enhanced loop flattening

Enhanced loop flattening strategically increases the iteration distance along certain control flow paths. In other words, some blocks execute in a later iteration than is strictly necessary. This iteration distance increase is like inserting "pipeline bubbles" that can reduce the RecII by increasing the iteration distance along inter-iteration feedback paths. For example, in Figure 4.5 imagine an extra "dead" iteration inserted between the last and second to last iterations. The highlighted inter-iteration feedback path would still have latency 5, but would have an iteration distance of 3, meaning the minimum II imposed by that path would be $1\frac{2}{3}$ instead of $2\frac{1}{2}$. However, increasing iteration distances by adding bubbles[4] also inflates the trip count for the flattened loop, so we want to increase iteration distances just enough to reduce the II.

A more detailed example is shown in Figure 4.7. Part (b) is a greedy flattening. It is usually possible for every code block (S1, S2, and S3) to execute in a single iteration of the flattened loop. This means that any feedback dependency cycle that goes through all blocks must complete in a single iteration, which in turn means that the II will have to be large enough to accommodate the longest cycle.

Part (c) of Figure 4.7 shows an alternative flattening that adds bubble iterations between the two inner loops. These extra iterations mean that to get from S1, though S2 and S3, and back to S1 will take at least 4 iterations in the flattened loop. Notice, however, that these bubble iterations only happen between the two inner loops. So, if the inner loops have reasonably high trip counts, adding bubble iterations between them will not increase the total trip count for the flattened loop by more than a few percent.

Part (d) of Figure 4.7 shows the almost fully flattened version of the loop nest. The syntax "(p) S" indicates that the statement S is executed in a predicated fashion with predicate p. This version of the code clearly illustrates that every piece of the original loop nest is executed in every iteration of the flattened loop—some statements are just predicated and therefore have no effect.

---

[4]In a well-scheduled pipeline, bubble iterations are not a time when no work is happening. Rather, bubble iterations allow extra time for slower infrequently executed chains of operations to complete.

(a)

```
do {
    S1;
    if (C2)
        S2;
    else
        S3;
    S4;
} while (C1);
```

(b)

```
p1 := true, p2_then := false, p2_else := false,
p2_else_1 := false, p2_else_2 := false, p3 := false;
while (true)
    if (p1)
        S1;
        p1 := false;
        if (C2)
            p2_then := true;
        else
            p2_else := true;
    if (p2_then)
        S2;
        p2_then := false;
        p3 := true;
    if (p2_else_2)
        p2_else_2 := false;
        p3 := true;
    if (p2_else_1)
        p2_else_1 := false
        p2_else_2 := true;
    if (p2_else)
        S3;
        p2_else := false
        p2_else_1 := true;
    if (p3)
        S4;
        p3 := false;
        if (C1)
            p1 := true;
        else
            break;
```

Figure 4.8: Another example of enhanced loop flattening. Here the original loop does not have any nesting or sequencing, but it does have an unbalanced if/then/else. Using enhanced loop flattening we can add extra "bubble" iterations to one side of the conditional, but not the other, which means that the II does not have to be set pessimistically high.

Figure 4.8 shows another use of enhanced loop flattening. In this case the original code does not have any nested loops, but it does have an unbalanced if/then/else. For this example, we assume that *C2* is heavily biased towards true, and S3 creates much longer inter-iteration feedback paths than S2. If we simply apply conventional if-conversion and software pipelining, the II will be high because of S3. However, as shown in part (b) of Figure 4.8, we can change the flattening to add bubble iterations only when *C2* is false. This increases the time available for inter-iteration chains that involve operations from S3, and hopefully allows the whole flattened loop to be scheduled at a lower II.

The enhanced loop flattening algorithm translates a control flow graph (CFG) repre-

sentation of a loop nest into a single dataflow graph (DFG) that can be scheduled with conventional software pipelining algorithms. The output DFG can be thought of as the body of a single loop where all code in the original loop nest executes in every iteration of the flattened loop. In order for the flattened loop to correctly implement the original loop nest, two issues must be dealt with:

- A set of predicates have to be used to guard operations with side-effects, so that the side-effects only happen when the logic of the original loop nest says they should.
- The flow of data from producer operations to consumer operations must be controlled correctly. For example, a use of variable `x` that appears after an if/then/else that causes different assignments to `x` needs to get the correct value, as dictated by the branch condition. We use trees of select operations in enhanced loop flattening to implement this data flow.

The novel contributions of our enhanced loop work are:

- a flexible framework for specifying iteration distances;
- a heuristic for deciding what iteration distances along specific control paths should be; and
- Iteration distance-sensitive algorithms for generating logic for predicates and select operations that heuristically minimize inter-iteration feedback paths.

4.3 Enhanced loop flattening implementation

There are two major subcomponents to enhanced loop flattening:

1. Given a control flow graph representation of a loop nest, assign an iteration distance to each edge.
2. Given a control flow graph with iteration distance annotations, generate the predicate logic and select operations to correctly implement the loop nest.

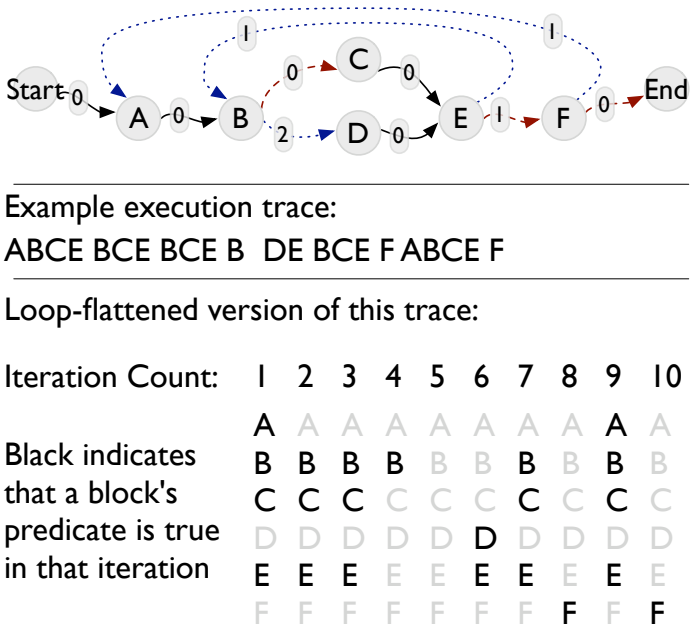


Figure 4.9: An example CFG with iteration-distance annotations, an example trace through that CFG, and a view of how that trace could execute in a loop-flattened version of the CFG. The specific iteration distance annotations were chosen for illustrative purposes, and do not necessarily represent the best choices for this graph.

4.3.1 Choosing iteration distances

Iteration distances are non-negative integer annotations on control flow graph edges. If there is a zero-distance edge between two blocks it means that if the predicate for the predecessor block is true in some iteration and it branches to the successor, the predicate for the successor will be true in the same iteration. If there is a d (> 0) distance edge between two blocks, it means that if the predicate for the predecessor block is true in iteration N and the edge condition is true, then the predicate for the successor block will be true in iteration $N + d$. We can think of this iteration distance as putting d bubbles in the software pipeline between the two blocks.

Figure 4.9 shows a control flow graph that has been annotated with iteration distances.

The enhanced loop flattening algorithm uses "intra-iteration" variants of many flow graph concepts like paths, dominance and topological order. A CFG edge with a distance of zero is intra-iteration, and an edge with non-zero distance is inter-iteration. The intuition for the intra-iteration variant of a flow graph concept is that you remove the inter-iteration edges from the graph and evaluate the flow graph concept with the remaining (intra-iteration) edges. Here are more formal definitions of the important relations reachablility, dominance and post-dominance.

x reach y	There is an intra-iteration path from x to y.
x dom y	Every intra-iteration path to y from either the start node or an inter-iteration edge includes x. "You can't get to y without going through x."
x pdom y	Every intra-iteration path from y to either the end node or an inter-iteration edge includes x. "You can't leave y without eventually going through x."

Figure 4.10: Intra- and inter-iteration flow graph concepts.

The "flattened trace" at the bottom shows how an example trace maps to iterations of the flattened loop. Notice that after the execution of block B in iteration 4 there are two "bubble iterations" before block D executes. This corresponds to the "2" annotation on the B-D edge in the CFG.

The only hard constraint on the selection of iteration distances for a CFG is that in every cyclic path there must be at least one edge with distance greater than zero. Zero-distance cycles imply executing the same block twice in the same iteration of the flattened loop, which does not make any sense (if loop unrolling is desired, it should be done before flattening).

Optimal selection of iteration distances involves minimizing the following formula for

total execution time of the flattened loop: $T = C_{flat} \times II_{flat} + L_{flat}$. Usually the latency is not significant, because it is only an extra cost during the prologue and epilogue of the entire loop nest. Ignoring latency leaves trip count (C) and initiation interval (II) as the variables to minimize. Increasing the iteration distance on an edge will increase C whenever the dynamic execution of the flattened loop follows that edge, but might lower the overall II if the critical-latency feedback path in the program involves values that flow through that CFG edge.

Two pieces of information should be known or estimated in order to optimize iteration distances: the relative execution frequencies of the CFG edges, and the criticality of the cycles. It is good to increase iteration distances on edges that participate in higher criticality feedback paths, because that is more likely to reduce the II. However, it is also good to keep iteration distances low on edges with high execution frequency, because that will inflate the trip count less.

Execution frequency information can be collected by profiling. Profiling for branch frequency has been implemented in many compilers, and is not conceptually complex, although it does require some additional infrastructure. The programmer has to provide a representative input set for the profiling run, and the intermediate representation has to be enriched so that the compiler can map the results of the profiling run back to its internal control flow graph. We have not implemented the necessary infrastructure for profiling, so we use a simple static heuristic to estimate execution frequency: edges inside of loops are assumed to execute more frequently than edges that enter or leave loops.

Loop flattening comes before scheduling in the compilation flow and needs estimates of the criticality of inter-iteration feedback chains, but the latencies of particular chains of operations (and thus their criticality) are not known until scheduling. Loop flattening could be performed multiple times with feedback from the scheduler, but again this requires more compiler infrastructure work. We currently use a simple criticality heuristic that is described below as part of the predicate and select implementation algorithm. We found this heuristic works well for our benchmark set.

4.3.2 Generating predicates and selects

This section covers the translation of a control flow graph that has been annotated with iteration distances into a dataflow graph. This dataflow graph represents the body of the single simple loop that is the output of loop flattening.

The only primitive DFG operation we use that is not common is the iteration delay (or just "delay"). Delays have an input, an output and a static parameter d. A delay operation's output on iteration $i + d$ is whatever its input was on iteration i. Delays may or may not be *initialized* with output values for the first d iterations. In hardware terms, delays are similar to chains of registers. In software terms, delays are like chains of d temporary variables that get "shifted" by one at the end of each iteration.

4.3.3 Basic blocks

Basic blocks are sequences of simple statements with no other control flow; once a basic block (sometimes just "block") starts executing, all its statements will execute. We use a static-single assignment (SSA) style representation for the basic blocks in our internal representation, which means that producer and consumer operations are directly connected to each other.

We do not strictly preserve the original ordering of operations from the source program, so operations that modify or read a stateful object require extra scheduling constraints to keep all operations on a particular object in program order. For example, without these constraints two writes to a single array could execute out of order, which could lead to incorrect results. Scheduling constraints are implemented as pseudo-dataflow dependencies that are removed after scheduling. Operations with side-effects are also individually predicated with a dynamic Boolean input to control whether they should perform their action on each iteration.

4.3.4 Predicates

Every basic block has a predicate that is true on iterations of the flattened loop when the block should execute and false on iterations when the block should not execute. Each block's

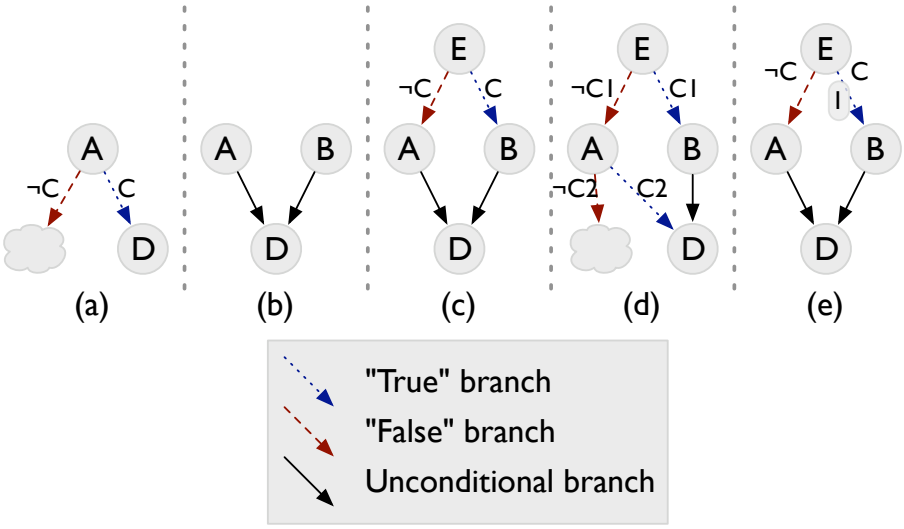


Figure 4.11: Different situations for generating a predicate for block D. In case (a), D's predicate is A's predicate ANDed with the branch condition. In case (b) D's predicate could be the OR of A's predicate and B's predicate. However, if the CFG looks like (c), D's predicate should just be identical to E's predicate. More complex patterns, like the irreducible graph in (d) require more complex logic. Non-zero iteration distances, like the edge from E to B in (e) make the logic more complex even for graphs that are structurally simple.

predicate is a function of the predicates of its predecessor blocks and the branch conditions on the edges between them.

We extend the notion of predicates to CFG edges by defining the predicate of an edge to be the predicate of its source block logically ANDed with the condition under which that block takes a branch along that edge. Edges whose source node has a single successor (i.e. an unconditional branch) have a predicate that is identical to the source node.

Figure 4.11 illustrates different CFG shapes and their relationship to predicate generation. Blocks that have a single predecessor (a) are the simplest case; the block's predicate is exactly the edge predicate of the block's single incoming edge. Blocks with multiple predecessors (b-e) can use the logical OR of their incoming edge predicates, however, this strategy

makes more complex predicates than is necessary. For example in case (c), the predicate for block D can simply be the same as block E. Irreducible CFG patterns, like case (d), and non-zero iteration distances, like case (e) require more complex predicate logic.

We will look at case (e) more closely, since it involves non-zero iteration distances. The predicate for block D in this case should be $(P_E \wedge \neg C) \vee \text{delay}_1(P_E \wedge C)$. In English, this means that block D's predicate should be true whenever in the current iteration block E's predicate is true and condition C is false, or in the previous iteration block E's predicate was true and condition C was true.

The simplest correct method for generating predicates is that each block's predicate is the logical disjunction of its incoming edge predicates (delayed appropriately). In the next section we show how to use the structure of the flow graph to generate more efficient predicate logic. However, it is also possible to use conventional logic synthesis tools like ESPRESSO[MSBSV93] to generate more efficient logic directly from the simple version. We have not yet quantitatively evaluated how well such an approach would work.

4.3.5 Predicate optimization

Our method for computing predicates is illustrated in Figure 4.12. It is similar to existing methods for if-conversion [CCF03], which in turn are based on the notion of control dependence from the program dependence graph (PDG) literature. To calculate the predicate for a CFG node x, we find its *intra-iteration* post-dominated region, the set $\{y | x \ \mathsf{pdom} \ y\}$. Intuitively, once one of the nodes in x's post-dominated region executes, we can be sure that x will execute in the current iteration.

The edges that cross from outside of x's post-dominated region into the region determine whether x will execute or not; these are called the control dependence edges for x. x's predicate is the logical OR of the predicates of the control dependence edges. However, some of these might be inter-iteration edges. For these edges, x's predicate does not depend on the current edge predicate, but the value of the edge predicate from d iterations in the past, where d is the iteration distance associated with the edge. We use delays to represent these inter-iteration connections. Delays in the predicate logic must be initialized to false

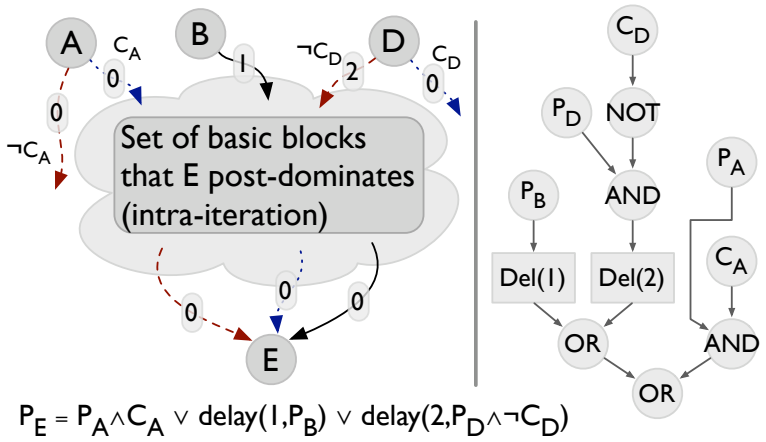


Figure 4.12: A piece of a control flow graph on the left, with the predicate logic for block E represented as a formula and a data flow graph. P_x means "predicate for block x", C_x means "branch condition for exiting block x", and $Del(N)$ means N-iteration delay.

to prevent spurious execution of predicated operations during the prologue.

The number of control dependence edges can be greater than two, which means that we have to choose how to best perform OR reductions. Normally balanced trees are the best approach to large reductions, but we found that it is most important to keep the critical paths as short as possible. We use a linear chain of binary OR operations, where the two least critical predicates are ORed together and the result ORed with the next most critical, and so on.

As noted earlier, our current implementation does not have support for the feedback from scheduling that is necessary to compute accurate criticality information. As an easily computed estimate of criticality, we use the smallest number of iterations it can take to get from the current block back to itself through a particular control dependence edge. For each control dependence edge, we count the smallest total iteration distance on any cyclic path backwards to the block for which we are generating a predicate. If there is no such path, we consider the distance for that edge to be infinite. Criticality is taken to be the inverse of this distance. This heuristic aims to keep latencies around the inner loops as low as possible.

4.3.6 Select operations

In order to process a basic block we need to know what the latest producer operation is for each variable that is live-in[5] to the block. Blocks that have a single predecessor are easy to handle: the definition for each variable is whatever it was at the end of the predecessor block. Blocks with multiple predecessors require extra work, as illustrated in Figure 4.13.

For basic blocks with multiple predecessors, there can be multiple producer operations for each variable. On each iteration the dynamic control flow determines which value should be seen by consumer operations. As suggested in Figure 4.13, there are two options: predicate the writes so that only one of them actually has an effect, or rename the variables and explicitly select between them based on the condition. Predicated register updates have been implemented in some VLIW architectures specifically to support this kind of transformation. However, even for conventional architectures this approach introduces performance challenges related to register renaming [WWK+01], and massively parallel architectures generally have no global register file at all. Therefore, we use explicit select operations in enhanced loop flattening.

Explicit select operations are similar to phi operators in static single assignment (SSA) compiler intermediate representations or "decoded multiplexors"[Bud03] in dynamic data-flow style representations. The difference is that our select operations, like hardware multiplexors, have an explicit control input whose value determines which input should be chosen.

The logic for inserting select operations is similar in some ways to predicate generation logic. However, there are some important differences as well. Figure 4.14 shows some simple cases of select logic generation. The most important difference between predicates and selects is that in every iteration the predicate for a block must be true or false, according to the control flow of the application and the chosen iteration distances. In contrast, in iterations where some particular block does not execute, it does not matter what the inputs to that block are. These *don't care* cases make it allowable to only use condition C1 in case (d). If the program follows the ¬C2 branch out of block A, it doesn't matter what

[5]Live-in variables are those that are read before they are reassigned.

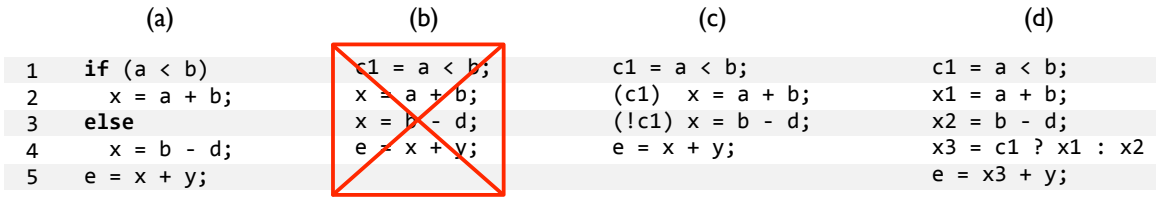


Figure 4.13: An illustration of the need for selects to control the flow of data in flattened code. (a) is the original code where some variable is updated in different ways along different control flow paths. Flattening results in code like (b) where both paths execute no matter what the condition evaluates to. Clearly this code does not work properly, because the second assignment to `x` always overwrites the first. To make the proper producer flow to consumers after the control flow reconvergence, we can predicate the updates (c) or add select operations and use static single assignment-style variable renaming (d).

the selects for block D do. Generating efficient select trees in more complex cases requires careful analysis of these don't care cases.

As we described for the predicates, it should be possible to use standard logic synthesis tools to optimize a simple specification of a block's select operations. The input to a block for some variable x is a function of the block's predecessors' edge predicates and data outputs for x. Figure 4.15 shows a truth table representation of the logic for a block with three predecessors. The important fact for efficient select generation is that if all the edge predicates are false, it does not matter which data value is selected.

Qualitatively, select optimization seems like a harder problem than predicate optimization, though we have not quantitatively evaluated this intuition yet. In order to do well on select logic generation, a logic synthesis tool would have to understand don't cares. In the following section we describe an approach to generating select logic that exploits the structure of the control flow graph to generate good logic quickly. This approach does not require a separate logic optimization step.

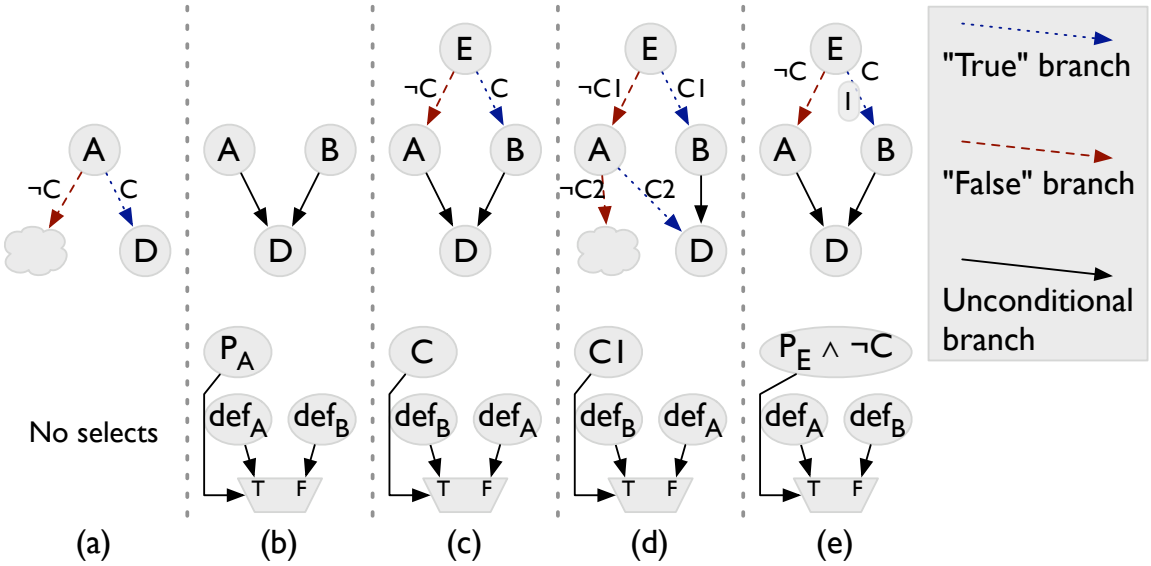


Figure 4.14: Along the top are the same snippets of control flow graphs from Figure 4.11. At the bottom are the selects that are required for block D variables that have different definitions in D's predecessor blocks. Case (b) illustrates the option of using a predecessor block's predicate as the control input for the selects. Case (c) shows that branch conditions, which are usually simpler than predicates, can be used instead. Case (d) illustrates the role of don't care cases; if C1 and C2 are both false, block D's execution predicate will be false, so it does not matter which input D's select operations choose. Case (e) hints at the complexity that non-zero iteration distances add to select logic generation; this issue is covered in more detail later.

EP_1	EP_2	EP_3	Out
T	F	F	D1
F	T	F	D2
F	F	T	D3
*	*	*	Don't care

Figure 4.15: A truth table representation of the simple select function for some block with three predecessors. The inputs to the function are the edge predicates along the three incoming edges, and placeholders for the three different data inputs. The output is defined to be one of the data inputs if one of the edge predicates is true and the others are false.

4.3.7 Select operation optimization

For acyclic reducible control flow graphs, efficient select logic generation is relatively simple. At every merge point in the CFG there is a corresponding branch, and the condition for that branch can be used to control the selects at the merge point. However, irreducible control flow and non-zero iteration distances make the process more complicated. Our algorithm for generating select trees for a particular basic block has two phases. First there is a backwards flow analysis that identifies the branches that have some influence on which input should be chosen. This analysis is intra-iteration; it stops at non-zero iteration distance CFG edges. The next phase considers the inter-iteration edges that can reach the block for which we are generating selects.

The algorithm for generating select operations for a *consumer* block labels CFG edges during the first phase with one of two kinds of labels:

- R_e, where R stands for "pRoducer"[6] and e is a reference to one of *consumer*'s incoming edges. R_e labels represent a direct input from one of *consumer*'s predecessors.
- The other kind of label is B stands for "branch", and it represents a choice between multiple producers, based on some condition. The syntax is $B(C, \langle e_1, l_1\rangle, \ldots, \langle e_n, l_n\rangle)$, where C is a reference to a specific branch condition, e_x is a reference to an edge and l_x is a label (Either R or B). The order of the edges matters in a B label. If C is a two-way branch, the first edge is "then" and the second is "else".

We will use these labels later as a schema or template for generating trees of select operations.

Algorithm for labeling all edges that can reach *consumer*:

1. Label each of *consumer*'s direct incoming edges, e, with R_e.
2. Visit the nodes that can reach *consumer* (intra-iteration) in reverse topological order. For each node n
   (a) let S_n be the set of edges that leave n and can reach *consumer*: $\{e \mid (source(e) = n) \wedge (e \text{ reach } consumer)\}$.

[6] We did not want to use P, to avoid confusion with predicates.

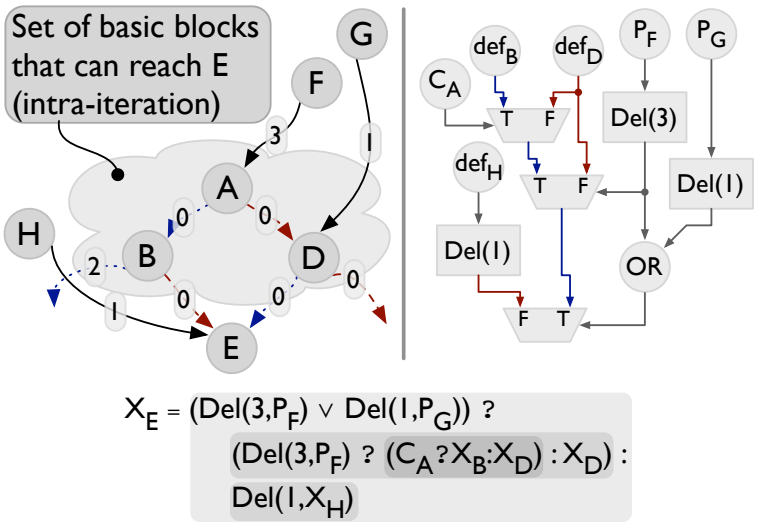


Figure 4.16: A piece of a control flow graph on the left, with the select logic for the incoming value of variable X to block E represented as a formula and a data flow graph. P, C, and *Del* have the same meaning as in Figure 4.12. X_y means the last definition of the variable X in block y. The "?:" expressions have the same meaning they do in C/C++/Java.

(b) If all edges in S_n have the same label, label all of n's incoming edges (intra- and inter-iteration) with that label.

(c) If not all of the labels are the same, label all incoming edges with $B(C, \langle e_1, l_1\rangle, \ldots, \langle e_n, l_n\rangle)$, where C is n's branch condition (n must end in a branch to have multiple outgoing edges), and the list of pairs are S_n, together with their respective labels.

⊠

After running this algorithm, all inter-iteration edges whose destination can reach *consumer* will have a label. We now have to consider two cases: either all these inter-iteration edge labels match or they do not. If all labels match we can generate a select tree for each variable by following the label with this intra-iteration select algorithm:

Algorithm intra-iteration select (variable x, label l)

1. if $l = R_e$ and e is an intra-iteration edge,
   return the last definition of x in e's source block.
2. if $l = R_e$ and e is an inter-iteration edge,
   return the last definition of x in e's source block, delayed by the iteration distance associated with e. These delays do not need to be initialized, because the value of the output does not matter unless the predecessor block is actually executed in an earlier iteration.
3. else $l = B(C, \langle e_1, l_1\rangle, \ldots, \langle e_n, l_n\rangle)$,
   build a select operation controlled by C with inputs determined by recursive invocations: intra-iteration select(x,l_1), ... intra-iteration select(x,l_n). If C is a multi-way branch, we decompose it into a chain of two-way selects, with the more critical inputs nearer the output of the chain. If the target architecture directly supports multi-way selection, that could be used as well. ⊠

There is a relatively important optimization that we apply in step 3 of the intra-iteration select algorithm. If all the definitions for x returned by the recursive calls to intra-iteration select are the same, there is no need for a select operation at all. Optimizing away selects with identical data inputs can be done as a later pass, but the number of unnecessary selects that would be created can be quite high, bloating the size of the intermediate representation.

Inter-iteration select control

Now we consider the general case where after running the labeling algorithm there are N inter-iteration edges that can reach the block and have *different* labels. We run the intra-iteration algorithm on each of the labels and then we need an additional layer of select operations to choose between these N options.

We use a heuristic for inter-iteration select logic generation that is illustrated in Figure 4.17. Like the predicate logic OR chains, we compute an estimate of the criticality of each of the paths, then make a chain of selects with the most critical select "closest" to the consumer and the least critical select "farthest". Select trees can be further optimized with more detailed criticality information, but that is beyond the scope of this dissertation.

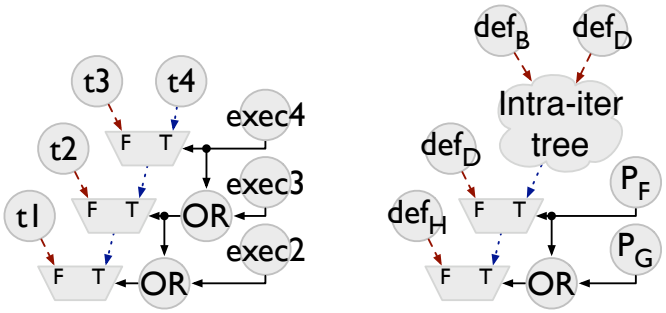


Figure 4.17: On the left is a generic select chain for the inter-iteration part of data flow control. "t_n" represents the mux tree generated from intra-iteration select label "l_n". t_1 is the most critical; t_4 the least. $exec_n$ is an expression that represents a predecessor block executing in an earlier iteration. Notice that the most critical "exec" is not part of the logic at all. On the right is a concrete example derived from Figure 4.16.

To control the select chain, we need some expression that represents a particular edge executing; we will call this $exec_e$. $exec_e$ can be the the edge predicate, but as we will see below it can also be some other predicate that is true on a superset of the iterations on which the edge predicate is true. The least critical select operation is controlled by $exec_e$ for the least critical edge; if $exec_e$ is true, select along the least critical path, if it is false, select along the next-least critical path. The next select in the chain is controlled by the OR of the two least critical $exec_e$ expressions, and so on.

This organization has the important benefit that $exec_e$ for the most critical edge is not part of the logic at all. Imagine a simple nested loop where the inner loop has two incoming edges: the loop-back and the external entry. The select operations for the inner loop can be controlled by the expression that represents that the loop is starting, so that they do not directly depend on the loop completion logic for the inner loop itself.

Now that we have a structure for the inter-iteration select logic, we need to choose how to implement the $exec_e$ expressions; the simplest correct way to do so is to directly use the edge predicate for e. This strategy is somewhat inefficient, because again there are many don't care cases to be exploited in select logic generation.

For each edge for which we need to compute an exec_e, consider the set of nodes that dominate its source node. The predicate for any of these dominators might be a good replacement for the edge's predicate, because we know that a dominator's predicate will be true on all iterations that the edge's predicate is true. The only problem is that the dominator might also reach one of the other inter-iteration edges that reaches *consumer*. So, for each edge we get its dominator set as well as the set of nodes that can reach all the other edges' sources. We subtract the reaching set from the dominator set and choose the highest (in the dominator tree sense) node that remains and use its predicate as exec_e for that edge. If there are no dominators left after the subtraction, we must use the edge's predicate. Whichever predicate we choose, we must then delay it by whatever the edge's iteration distance is.

4.3.8 Scheduling, placement and routing

After flattening is complete, the resulting loop can be scheduled by conventional software pipelining algorithms. Note that there is nothing special about the predicate and select operations; they should get scheduled and have their outputs pipelined, just like any other operations.

For applications with non-trivial loop nesting, the predicate logic and select trees created by our flattening method can be sufficiently complex that we expect targeting conventional processors would not work very well. In particular a large number of predicate values have to be maintained and pipelined. This would create severe register pressure and would be an inefficient use of wide registers for Boolean values. In [MHM+95] the authors observe a similar problem with predication, and propose architectural features to address the problem.

Our current target architectures are FPGA-like in the sense that they have large amounts of compute resources compared to conventional VLIWs, support deep pipelining of data values well, and support Boolean computation efficiently. Because these architectures are spatial, our back-end must perform not only the scheduling and resource allocation of conventional software pipelining, but also temporospatial placement and routing, like that described in [MVV+02, FCVE+09].

4.3.9 Complete enhanced loop flattening algorithm

Enhanced Loop Flattening(input *cfg*)

- Use profiling information and/or scheduling feedback to assign iteration distances to edges in *cfg*. For now we use the heuristic that all back-edges and edges that enter or leave a loop are assigned iteration distance 1; all others are assigned 0.
- Visit each basic block, *bb*, in intra-iteration topological order (that is, all intra-iteration predecessors of a block will be visited before that block.)
  - Create a predicate, P_{bb}, for *bb*:
    - Find *PD*, the set of *cfg* nodes that *bb* intra-iteration post-dominates.
    - Let E_C be the set of edges whose destination is in *PD* and whose source is not in *PD*.
    - Sort E_C by the criticality of the predicate associated with each edge.
    - Make a linear chain of binary OR operations with the predicates from E_C as the inputs, where the most critical predicate is closest to the final output and the least critical is farthest. For inter-iteration edges, insert a delay operation with the delay parameter set to the iteration distance on the edge.
  - Create a select schema for *bb*. The procedure for building a select schema is described in Section 4.3.7, and is too complex to repeat here.
  - For each variable x
    - Build a select tree for x based on the schema. If at any point both data inputs to a select are the same, eliminate the select.
    - Add an entry to *bb*'s input symbol table for variable x.
  - Do an SSA-style translation of the (straight-line) code in *bb* into DFG operations, predicating side-effecting operations with P_{bb}.
  - Update the symbol table associated with each of *bb*'s outgoing edges to the final symbol table from this block's translation.

Application	Abbr.
2D convolution	conv
CORDIC	cord
Dense matrix multiplication	mm
Event detection	ed
Finite impulse response (few coefficients)	firs
Finite impulse response (many coefficients)	firl
K-means clustering	km
Matched filter	mf
Motion estimation for video compression	me
Smith-Waterman	sw

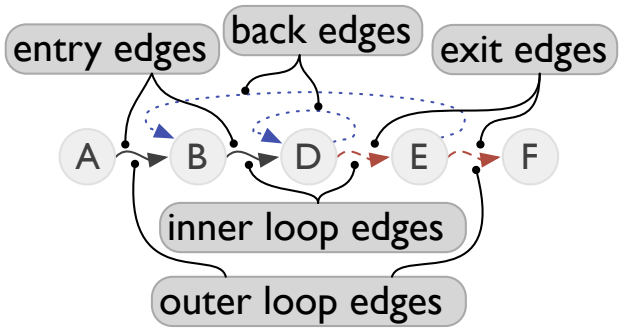
Figure 4.18: Benchmarks used in our evaluation of enhanced loop flattening.

4.4 Evaluation

In the context of C-like languages for parallel accelerators, the most important advantage of enhanced loop flattening over plain software pipelining is that it works on applications that have more complex control flow. Existing systems for translating from sequential languages to highly fine-grained parallel architectures, like Impulse-C [PT05] and StreamC/KernelC [KRD+03] force the programmer to settle for inner-loop-only pipelining (ILOP) or do the flattening transformations by hand, which is hard to get right and results in messy, unmaintainable code.

To quantify the benefits of enhanced loop flattening, we compare the run times of a set of benchmarks compiled with enhanced loop flattening in a number of different experimental configurations. The benchmark applications we used are listed in Figure 4.18. Three of the applications (cordic, firs, and ed) have only a single loop, so there is no difference between enhanced loop flattening and conventional software pipelining. Thus, we do not report results for these applications.

Our target architecture is a simulated accelerator with 200 ALUs arranged in clusters



Entry	Exit	Abbr.
All	All	A-A
All	Inner	A-I
All	None	A-N
Inner	All	I-A
Inner	Inner	I-I
Inner	None	I-N
None	All	N-A
None	Inner	N-I
None	None	N-N

Figure 4.19: The family of static iteration distance heuristics utilize the loop structure of the program. Particular heuristics in the family put different iteration distances on edges based on whether they are entry or exit edges, and whether they are entering/exiting an inner or outer loop. For example, the "AN" version would put extra iteration distance on edges A-B and B-D, but not D-E and E-F.

with a grid interconnect, like that described in [VWC+09].

All results are normalized to the ILOP implementation. We implement ILOP in our infrastructure by adding extra scheduling constraints that prevent the overlap of operations from different blocks, except the overlap of inner loop operations with each other. We then add enough iteration distance on outer loop connections to cover the latency of those blocks. This effectively prevents epilogue and prologue overlapping, except for inner loops. The other point of comparison is conventional loop flattening, which we call greedy because blocks are scheduled in the earliest possible iteration of the flattened loop.

Enhanced loop flattening provides a flexible framework for setting iteration distances, but leaves open exactly what those distances should be. We implemented and evaluated a family of heuristics for setting iteration distances based on the loop structure of the program. We use the algorithm from [SGL96] to identify the CFG nodes that participate in all loops, including arbitrary nesting of reducible and irreducible loops. Here we mean loop in the programming language sense, not the graph theoretic sense, so a simple diamond-shaped (or hammock-shaped) CFG with a back-edge from the bottom to the top is a single loop, not two.

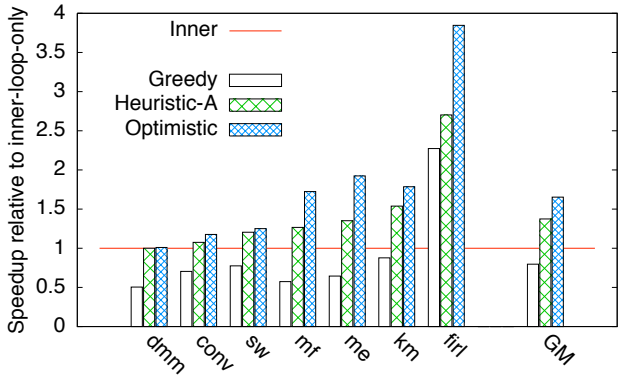


Figure 4.20: Normalized speedups for a suite of applications compiled with enhanced loop flattening. The baseline is inner-loop only pipelining (ILOP). "Greedy" is conventional loop flattening with non-zero iteration distances on back edges only. "Heuristic-A" is our enhanced loop flattening with an iteration distance of 1 on all loop entry and exit edges. "Optimistic" is a bound on how well the iteration distances can be selected. "GM" is geometric mean.

By default, control flow edges are intra-iteration (have iteration distance zero). Once we have identified the loops, we give all all back-edges iteration distance one. Back-edges can be identified in the conventional way with a depth-first traversal of the nodes in each loop. After this, different members of the heuristic family assign one to different additional edges. Figure 4.19 illustrates the different kinds of edges. In this setup, we can choose to increase the distance on all loop entry edges, only inner loop entry edges or no loop entry edges. We have the same set of choices for loop exit edges. If we choose to add distance to no entry or exit edges, we are back to the greedy loop flattening case.

Figure 4.20 shows the results for the 7 benchmarks that have more than a single loop in the kernel. The "Heuristic-A" bars show the runtimes when the iteration distances are set to one on all loop entry and exit edges ("A-A" from Figure 4.19). We see that the greedy flattening is almost always worse than ILOP. The enhanced loop flattened implementations with our simple heuristic for setting iteration distances are all at least as fast as ILOP, and some are substantially faster.

Greedy loop flattening performs poorly because the low iteration distances cause the minimum initiation intervals to be higher than the minima for just the inner loops. Table 4.1 shows the performance data broken up into initiation interval and trip count. High initiation intervals have a very negative impact on performance because they cause every iteration of the loop to take more time than is necessary.

The heuristic enhanced loop flattening performs much better than greedy flattening, even though the trip counts are somewhat higher, because the initiation intervals are much lower. The "Optimistic" bars in Figure 4.20 show the speedups we would see if we could get the lower trip counts of greedy flattening and the lower IIs of enhanced flattening at the same time. This represents an optimistic bound on the performance improvement that could not be exceeded by choosing better iteration distances.

The application characteristics that determine the effectiveness of enhanced loop flattening relative to ILOP are the latency of the inner loop(s) and the trip counts of the inner loops. Enhanced loop flattening allows useful work to be done during the fill and drain periods of the inner loops; the latency determines how long each fill/drain period is, and the lower the trip counts of the inner loops, the less steady-state execution there is to amortize the fill/drain inefficiency over.

In our benchmark set, dense matrix multiplication (dmm) shows very little performance difference between enhanced loop flattening and ILOP. The reason for this is that we use a SUMMA-style algorithm where the inner loop is essentially many parallel multiply-accumulate loops. This means that the latency of the inner loop is very low and the benefit of pipelining around the outer loops is small as a consequence. At the other extreme, our banked FIR filter implementation (firl) shows a large performance difference, because there is a very long-latency chain of additions in the inner loop.

As a final note on the inner loop only experiments, we believe that our method is quite optimistic about the performance in the ILOP case. Depending on how the outer loop code is compiled and what the target architecture is, there might be significant additional overhead associated with entering and leaving inner loops.

Table 4.1 shows the performance results broken up into initiation interval and trip count. We also include data for enhanced loop flattening with the iteration distance heuristic set

App	Initiation Interval				Trip Count			Speedup		
	ILOP	Grdy.	I-I	A-A	Grdy.	I-I	A-A	Grdy.	I-I	A-A
mm	4	2.0	1	1	0.99	1.00	1.00	0.51	1.00	1.00
conv	3	1.67	1.33	1	0.85	0.93	0.93	0.70	0.81	1.08
sw	5	1.6	1	1	0.80	0.83	0.83	0.78	1.20	1.20
mf	3	3.0	1	1	0.56	0.74	0.79	0.60	1.35	1.27
me	4	3.0	1	1	0.52	0.69	0.74	0.64	1.45	1.35
km	4	2.0	1	1	0.57	0.65	0.65	0.88	1.54	1.54
firl	3	1.67	1	1	0.26	0.37	0.37	2.30	2.70	2.70
GM		1.66	1.03	1	0.71	0.79	0.80	0.80	1.34	1.37

Table 4.1: Performance data broken up into initiation interval and trip count. The first data column shows IIs for ILOP in units of clock cycles. All other data are normalized to the ILOP case. In addition to the greedy and heuristic with non-zero iteration distances on *all* entries and exits (A-A), we also show results for the static heuristic with non-zero iteration distances on *only* the inner loop entries and exits (I-I).

to add distance to entry and exit edges for only the inner loops (I-I). The first column shows the natural initiation intervals of the inner loops of the applications, and the rest of the data is all normalized to the ILOP values. As shown in the Trip Count section of the table, adding more iteration distance (A-A vs. I-I) increases the trip counts slightly. The A-A configuration achieves a slightly higher average performance than I-I because there is one application that is not able to get back down to the minimum II in the I-I configuration.

Our final experiments examined the value of the more sophisticated dominator-based inter-iteration select control mechanism described at the end of Section 4.3.7 compared to directly using the edge predicate of the inter-iteration edge. This optimization has no effect on trip count and only a small effect on the latency of critical feedback loops. The effect was only large enough to actually reduce the II in a very few cases, so we looked at a different quality metric that is a more precise measure of the "slack" in the inter-iteration feedback paths. For every operation that participates in a feedback path we compute how much additional latency that operation would need to have in order to make it II-critical (i.e.

App	A-A	A-I	I-I	A-N	N-N
mm	1.040	1.026	1.049	1.0	1.0
conv	1.071	1.100	0.434[7]	1.0	1.0
sw	1.034	1.034	1.034	1.035	1.016
me	1.0	1.0	1.001	1.003	1.0
km	1.124	1.041	1.100	1.114	1.015
firl	1.0	1.0	1.0	1.0	1.0
GM	1.044	1.033	0.896	1.025	1.005

Table 4.2: Ratio of the average slack in inter-iteration feedback paths when the more sophisticated dominator-based method was used over the simpler direct predecessor method. The columns represent different iteration distance insertion heuristics, as described in Figure 4.19. The dominators method generally results in slightly greater slack. The only exception is conv in the "I-I" case, where the dominator method actually achieved a lower initiation interval (3 vs. 4).

any more latency would cause the minimum RecII to increase). We use the table method from [Rau94b] to compute these numbers. We found that using the dominator-based select logic led to noticeably higher average slack, which means that those implementations were fractionally closer to achieving a lower II.

Table 4.2 shows the slack data for our benchmark set. The difference between using the dominator method versus directly using the inter-iteration edge's predicate is not huge. However, it can only improve performance, and we believe that in applications with more complex control flow and iteration distances it may have a larger impact.

4.5 Discussion

Loop flattening and its acyclic cousin if-conversion share an important performance challenge, which is that all operations execute on every iteration, whether their results are used

[7]The slack in this case is much lower because the application compiled with a lower initiation interval when using the dominator-based select logic.

or discarded. In loop nests with lots of infrequently executed operations, this can lead to a lot of wasted execution resources. In the future we plan to address this problem by exploring ways that operations that are guaranteed to never execute in the same iteration of the flattened loop can share physical resources. This problem was addressed in the acyclic case by the authors of [SMDL03] (who mostly ignored spatial distribution issues). We believe that extending those results to incorporate spatial distribution and iteration distances is an interesting and tractable challenge.

To facilitate this sharing between operations that execute in mutually exclusive conditions, we build a *control dependence graph* (CDG)[CFS90], which keeps track of the conditions under which each part of the DFG should actually execute. We are also considering a modified CDG that represents iteration distances between different branches. This extended CDG should support sharing of resources between, for example, operations in different inner loops in the same loop nest.

Another approach to avoid executing unnecessary operations is "reverse if-conversion" or "control flow regeneration" [WHSB92, WMHR93, MJ02]. The idea is that after scheduling, true control flow can be used instead of predication to control execution and data flow. The problem with this approach is that pipelined scheduling does not respect basic block boundaries, and operations from different branches get spread out in time, which makes the control flow much more complex. The additional control flow complexity grows exponentially with the depth of pipelining, which makes this approach inefficient for most applications.

4.5.1 Unbalanced diamonds

As suggested earlier, enhanced loop flattening offers an elegant solution to the classic unbalanced diamond control flow problem, where one side of a conditional statement is "fast" and executed frequently, and one side is "slow" and executed infrequently. With conventional if-conversion followed by software pipelining, the compiler is forced to choose an initiation interval high enough to accommodate execution of the slow branch in consecutive iterations. With enhanced loop flattening, we can increase the iteration distance on the slow

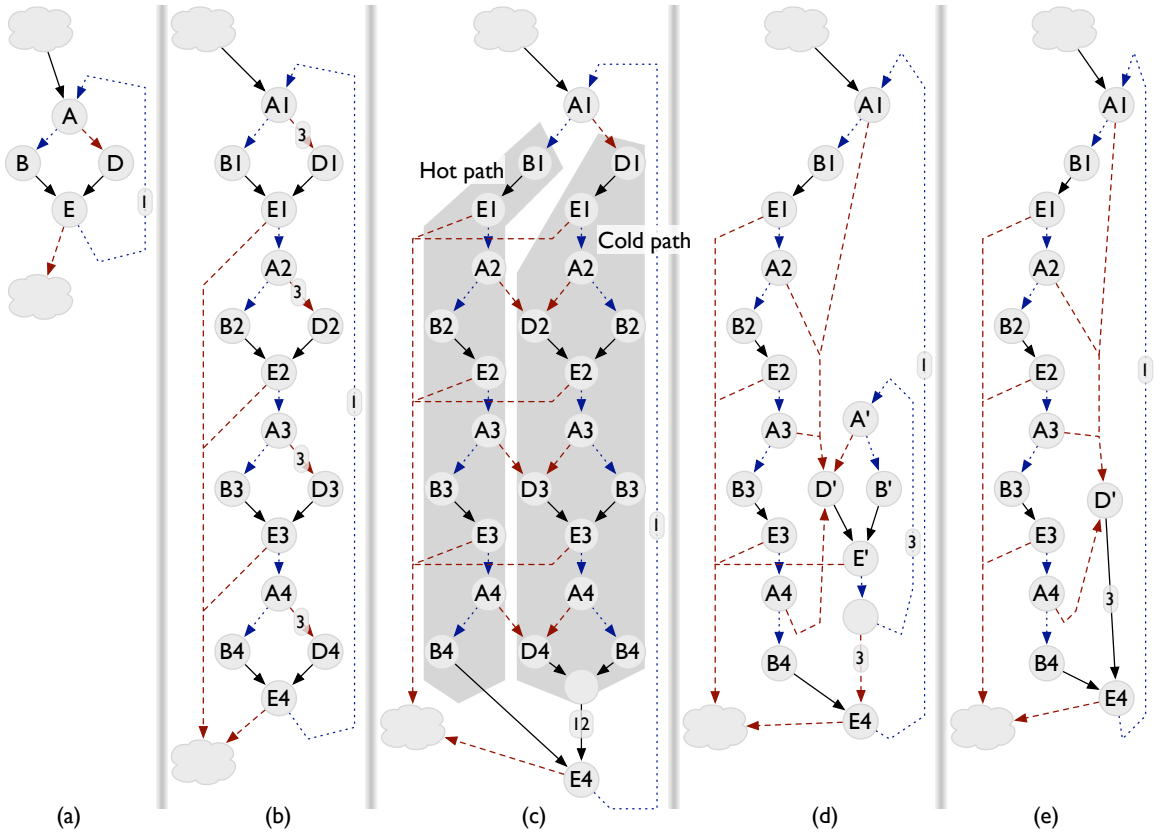


Figure 4.21: Trace scheduling in the enhanced loop flattening framework. (a) is an inner loop with one internal conditional branch. For this example we will assume that B executes much more frequently than D, and D has some operations that create a long latency feedback path. In order to increase parallelism we unroll the loop to get the CFG in (b). For reasons explained in the main text, the trace version (c) can be more efficient than (b). (d) shows the idea of "rerolling" the cold path to avoid unnecessary code duplication. (e) is another variation that has the interesting property that the set of dynamic iterations executed by a particular static copy of the hot path depends on the dynamic control flow. In all the other variations, B2, for example, will always execute iterations 2, 6, 10, ...

side, which will increase the number of iterations it takes to follow that path, but reduce the II.

Figure 4.21 gives an example of the flexibility that the enhanced loop flattening framework provides. In part (a) is a CFG for a simple loop with an unbalanced "if" (B is the fast, frequently executed block). First we unroll the loop by a factor of 4 (b) to increase the available parallelism. We have also increased the iteration distance along the A-D edges to improve the II. Adding the iteration distance creates an interesting new problem: if all the A-D iteration distances were 0, the predicate for E4 would be exactly the same as the predicate for A1. However, with the higher iteration distances, the predicate for A2 depends on the branch out of A1, the predicate for A3 depends on the branch out of A2, and so on. This chain of dependencies significantly complicates the predicate logic in a way that is very likely to be on the critical path. Adding the iteration distances solves one problem, but creates more complex predicate logic.

We can use the idea of trace scheduling [Fis81] to get the more efficient implementation in (c). As long as the branches all go in the B direction, execution stays on the hot path and there is no reconvergence from the cold path. Within the cold path itself we do not need extra iteration distance for each D block, just one large distance where the cold path rejoins the hot path. Reconvergence in the control flow graph of paths with different iteration distances create complex predicate and select logic, and the trace implementation (c) has fewer such reconvergence points than the implementation in (b).

Parts (d) and (e) show further possible optimizations of the control flow for this loop. In (d) we show "re-rolling" the cold path,[8] which can help save instruction memory. For the re-rolled implementation to work correctly, the compiler would have to add a counter variable that would be set differently by A1, A2, A3, and A4. If the unrolled copies of the loop body are truly identical, we can use the cold path short-cut in (e).

We have not fully implemented these trace-based CFG optimizations. The reason for describing them here is to give a sense for the kinds of optimizations that enhanced loop flattening makes easier. The CFGs in (c), (d), and (e) are irreducible, but that is not prob-

[8]Loop re-rolling is a code transformation that is exactly the reverse of loop unrolling. The term is used in the compiler community, but the transformation is not as widely used as unrolling.

lematic for enhanced loop flattening. Also, in working on a preliminary implementation of trace-based optimization, we found that the predicate and select logic increased in complexity somewhat. An efficient implementation would need to use detailed criticality feedback from the scheduler and more sophisticated logic minimization algorithms.

4.6 Summary

In this chapter we presented enhanced loop flattening a compiler framework for software pipelining nested loops with arbitrarily complex static control flow. The ability to overlap epilogues and prologues seamlessly around inner and outer loops is most beneficial when the pipelining is very deep[9] and the trip counts of the inner loops are not extremely high. Enhanced loop flattening provides a flexible mechanism for increasing the iteration distance along specific control paths, which can be used to balance the competing performance factors of trip count and initiation interval of the flattened loop.

In order to avoid unnecessary inter-iteration feedback paths, we proposed algorithms for predicate and select operation generation that use "intra-iteration" variants of the classic flow graph relations reachability, dominance and post-dominance and heuristics for minimizing inner loop latencies. Our implementation and the experiments we performed with it showed that even for relatively simple loop nests, pipelining the flattened loop has a better combination of trip count and initiation interval than inner loop only pipelining. We also showed that greedy loop flattening–with non-zero iteration distances on back edges only–creates long inter-iteration data flow feedback paths. Increasing iteration distances on less frequently executed control paths can improve performance by decreasing the initiation interval.

Automatic software pipelining is a necessary optimization for C-like language compilers for accelerators. Existing compilers support pipelining only loops that fit restrictive patterns. Most commonly, inner loop only pipelining is supported, which leads to badly suboptimal performance for applications that relatively long chains of dependent operations and relatively low inner loop trip counts. For such applications it is currently common for

[9]Deep pipelining can be seen as a large number of iterations overlapped in the steady state, or a high ratio of the latency of an iteration to the initiation interval.

programmers to hand-optimize their code to achieve roughly the same effect as enhanced loop flattening. Such hand optimizations obscure the main logic of the application and are hard to maintain. With enhanced loop flattening, programmers can use a natural coding style and get the performance benefit of pipelining around all kinds of control flow.

Chapter 5

A SHORT SURVEY OF TUNING

Tuning is the process of adapting a program to a particular target architecture or class of target architectures. Automatic and semi-automatic tuning has been a topic of interest in the high performance computing community for many years, and tuning for embedded systems and even general purpose processors has been growing in importance recently. In fact, according to [CDG+06], the gap between peak performance and what is typically achieved by conventional compiler optimization has widened, not shrunk, as the technological arms race between architects and compiler writers has progressed over the last few decades.

For parallel coprocessor accelerators tuning is at least as important as it is for conventional processors. Accelerators have explicitly managed resources, like distributed memories and non-uniform local networks, that applications must use well to achieve good performance. Accelerators present the additional challenge for automatic tuning that there are relatively few graceful fallback mechanisms built into the architectures. If a particular configuration of a program needs to use more local memory than is available, the program will simply fail to compile or run properly. Thus, tuning for accelerators combines searching for high values of a quality function while satisfying a number of resource constraints, whereas conventional approaches to tuning focus on just the quality function. Quality-only methods can be adapted by giving a default "very bad" quality to configurations that violate some constraint. However, as we will see in the next chapter, this strategy does not produce good results.

Chapter 6 presents a new probability-based method for tuning applications with a more sophisticated treatment of constraints. To put that work in perspective, this chapter is a survey of existing approaches to automatic and semi-automatic tuning. Automatically adapting programs to architectures in a way that maximizes performance is a challenging problem that has inspired a diverse set of solutions.

5.0.1 Tuning and portability

Auto-tuning is useful for getting a particular application to perform well on a particular machine. A related but distinct concern is portability, and specifically *performance portability*: it is hugely valuable to be able to recompile/retune a source program to an efficient implementation on a new architecture with minimal additional engineering effort. Performance portability not only allows programmers to target a wider set of architectures more easily, it also frees system designers to innovate without requiring laborious porting of a large set of applications.

One of the important differences between C and less abstract languages is that a program developed in C on machine X should be a simple recompile away from running reasonably well on machine Y, as long as the program is written in a portable style. Not only do general purpose C programs work correctly when recompiled on different machines, but they also generally perform well even if the architectures in question have very different resources. This kind of performance portability is an important part of the meaning of C-level.

Even in the world of C and conventional processors achieving performance portability has become more challenging as architectures have become more complex. Tuning of performance-critical libraries, like linear algebra routines and fast Fourier transforms, has become important. The architectural challenges that motivate work like this are more pronounced for accelerators and large parallel machines.

5.0.2 Outline

The two extremes of the tuning space are fully manual tuning, where the programmer explicitly encodes a particular tuning configuration into the source program, and fully automatic one-shot compilation, where the compiler uses some predictive models of performance to make tuning decisions. Fully manual tuning has the drawback of requiring huge human effort. Fully automatic compilation has been shown to produce badly suboptimal code in a number of contexts. The systems covered in this survey aim to strike a better balance between the extremes: less effort than fully manual and better performance than fully automatic. They can be distinguished based on how the following questions are answered:

- What is being tuned?
  - A specific library designed explicitly for being tuned
  - A broad class of programs
  - Not specific programs at all, but rather the compiler infrastructure itself
- How is the space of possible configurations defined?
  - An engineer explicitly defines the space of configurations
  - The configuration space is hard-coded into the system
  - The space is inferred by the tuning system
- How is the space of possible configurations explored?
  - Predictive models.
  - General heuristic searching (exhaustive, binary search, genetic, ...)
  - Domain-specific hybrid.

5.1 Background

Conventional optimizing compilers transform programs with the intension of improving the "quality" of the program, by some metric like execution time or energy consumed. They use models to estimate whether a particular transformation will improve the quality of a program or reduce it. However, modern architectures are so complex and unpredictable that creating precise models is challenging. As a consequence, conventional compilers produce code that is badly suboptimal in many contexts.

The failure of conventional compilers motivated investigation of a wide range of empirical approaches to compilation. The unifying idea behind empirical optimization is that we can get information about hard-to-model characteristics of a program running on an architecture by simply running it and seeing what happens. Typically, empirical compilers try running several variations of a program to see which works best. Empirical compilers and tuners do not necessarily discard the kinds of models used by conventional compilers, but rather use empirical feedback as additional input used to make optimization decisions.

The primary advantage that empirical approaches have is that they can use concrete performance information that is unavailable to conventional compilers. The primary disad-

vantage is that compilation can take a substantially longer time. Secondary disadvantages are that the compilation infrastructure is more complicated in some cases, and it is harder to know *why* an empirical compiler produces good or bad results, and thus how to improve it.

5.1.1 Complexity and unpredictability

Tuning is important because modern computers are complex and unpredictable. Most mainstream modern architectures include features like caches, branch predictors and out-of-order execution that make the runtime performance characteristics of a program less predictable. For tuning systems an important question is whether to treat architectures as complex but predictable, in which case more sophisticated models could accurately predict performance, or so unpredictable that only rough estimates of the performance of a configuration can be made before it is tested. The distinction between complexity and unpredictability is not black and white. For example, caches can be treated as totally unpredictable, making the latency of any given memory reference unknown, within some range. However, if the characteristics of a particular cache are known, models can be built which transform the unpredictability into complexity. In general, the harder it is to build a model of some feature, the more likely it is that empirical optimization will have some advantage over model-based optimization.

Many processors found in embedded systems, like DSPs (Digital Signal Processors), are more predictable but also more complex than conventional processors. DSPs tend to use less aggressive prediction and speculation, and expose more byzantine micro-architectural detail, like partitioned register files and local explicitly managed scratchpad memories.

Multiprocessors, which have recently moved into commodity computers and promise to offer greater degrees of parallelism as circuit technologies continue to scale, introduce significant new sources of complexity and unpredictability. For example, dynamically routed networks make the communication latency between processors depend on the communication patterns of all the processors sharing the network, and shared caches make the memory latencies experienced by a given processor dependent on the memory access patterns and

timing of other processors.

5.1.2 Transformations

Now we take a look at the kinds of techniques that compilers and tuners can use to deal with the complexity and unpredictability of modern computers. The program transformations made by optimizing compilers can be placed in three categories:

- Necessary transformations
- Classical optimizations
- Restructuring optimizations

Necessary transformations are tasks like instruction selection, register allocation, and instruction scheduling that essentially all modern compilers perform in some way. Classical optimizations, like dead code elimination and common subexpression elimination, are usually relatively local in scope, almost always have a positive impact on the quality of the code produced, and rarely have unpredictable negative side-effects. Restructuring optimizations include loop transformations like unrolling, strip-mining, blocking, interchange and software pipelining. A simple example, taken from [BGS94], is presented in Figure 5.1. Restructuring optimizations can dramatically improve the performance of programs, but can also have a negative impact, for example by increasing the code size or creating bad cache effects. Many restructuring optimizations take some extra parameter(s), such as the unrolling factor for loop unrolling. Optimizations with parameters increase the size of the search space much more than optimizations that are simply on or off.

All the existing work described in this chapter takes as a baseline a compiler that does at least a reasonably good job with the necessary transformations and classical optimizations. Some of the work focuses more on using empirical feedback to do a better job with these transformations, and some focusses on enabling or enhancing more dramatic restructuring optimizations.

```
do i=2, n-1
  a[i] = a[i] + a[i-1] * a[i+1]
end do
```

```
do i=2, n-2, 2
  a[i] = a[i] + a[i-1] * a[i+1]
  a[i+1] = a[i+1] + a[i] * a[i+2]
end do

if (mod(n-2,2) = 1) then
  a[n-1] = a[n-1] + a[n-2] * a[n]
end if
```

Figure 5.1: Simple loop unrolling example (in Fortran) taken from [BGS94]. The loop on the right does twice as much work per iteration and has half as many iterations. Unrolling is a program transformation that can have both positive and negative effects on program performance, depending on the program, the degree of unrolling, and the target architecture.

5.2 Improving mostly conventional compilers

The first family of compilers we will consider use empirical feedback to improve the quality of all or nearly all programs for a particular target architecture. There are three subfamilies that we will look at, in roughly increasing order of complexity: profiling compilers, empirically tuned compilers and empirical restructuring compilers. The fact that these compilers can improve the quality of a wide range of programs with relatively little change in the programming environment is clearly very attractive. The main shortcoming of this approach, relative to the library approach described in the next section, is that the program quality improvements achieved are modest—on the order of tens of percent, as opposed to more than a factor of two.

5.2.1 Profiling compilers

Profiling compilers [CMmWH91, CBM+93] are the simplest of the compilers that use empirical feedback, and are also the most widely used. The usage flow of profiling compilers is:

1. Compile source in profiling mode to produce a version of the program that will gather and save statistics as it runs.
2. Run the generated program on one or more representative input data sets.

3. Compile again with the gathered statistics as an extra input, to produce a final compiled program.

The two most useful pieces of profile information most compilers use are the execution frequency of various paths, and data dependence information that is either impossible to know or difficult to discover without runtime information (for example, whether two pointers are expected to alias).

Profiling compilers typically use profile data to tune necessary transformations and classical optimizations, but not restructuring transformations. A single compile-run cycle of a program generally does not provide enough information to prune the enormous search space of possible program restructurings substantially. Of course, profiling compilers can still use primarily model-based approaches for restructuring.

The performance improvement achieved in one of the most heavily cited works on enhancing conventional compiler optimizations with profile information ([CMmWH91]) on a broad suite of C benchmarks was 15% on average. These results clearly demonstrate the strengths and weaknesses of profiling compilers: they require relatively little change in programming environment, relatively little extra compilation time and can be applied to essentially all existing programs, but achieve only modest performance gains.

5.2.2 Empirically tuned compilers

The second group of compilers we will consider do not empirically adjust to particular programs, but use extensive empirical feedback to tune the compiler itself [CSS99, CST02, KZM+03, KHH+04]. This tuning can be used to make the compiler simply better for all programs, or to tune it to particular application domains.

All mainstream compilers attempt to heuristically find good solutions to problems for which finding optimal solutions is intractable, like register allocation and instruction scheduling. These heuristics can be quite complex, and ideally combine information about both the program being compiled and the target architecture in sophisticated ways.

Stephenson, et al. [SAMO03, Ste06] have developed a method, based on machine learning, to tune compiler heuristics automatically to particular systems. Many heuristics use

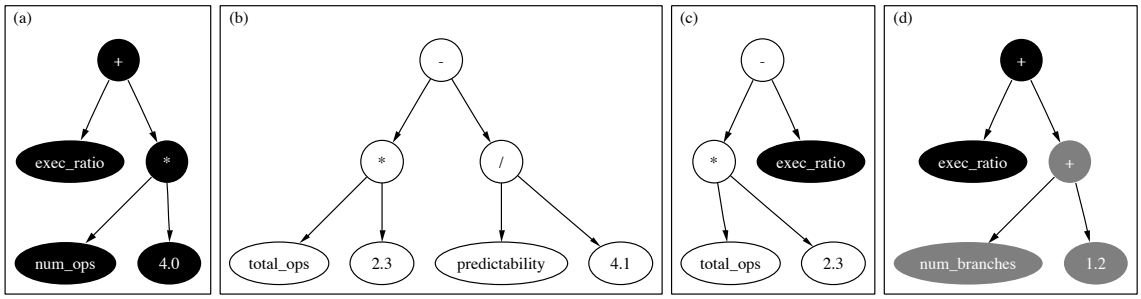


Figure 5.2: Example functions for use in compiler heuristics. These kinds of functions are used by compilers to make decisions about register allocation and code layout for branch optimization. Graphic borrowed from [SAMO03], which proposes a genetic approach to optimization, as suggested by the mixture of (a) and (b) to produce (c). (black = from example formula (a); white = from example formula (b); gray = a "mutation" from neither formula.)

simple mathematical combinations of several pieces of data to make decisions. For example, when a register allocation algorithm is deciding which variable to spill to memory, it might combine the number of times each variable is accessed and the size of each variable's live range to come up with a metric for which is the "best" variable to spill. There is a considerable amount of art and black magic involved in tuning these heuristic functions. Furthermore, it is extremely hard to know how close a particular function is to optimal, and thus whether it is even worthwhile to spend more programmer effort on optimizing it.

"Meta optimization" injects a measure of science and engineering into the process of tuning compiler heuristics. During the optimization process, the compiler randomly selects heuristic functions of the form seen in Figure 5.2. The functions can use any of the pieces of information about the program or the architecture that the compiler collects. It then compiles and runs some set of benchmarks and collects performance data. The heuristic functions are then randomly recombined, with pieces of functions that lead to high performance getting higher priority than functions that lead to low performance.

The amount of time needed to tune a compiler is very large, because several iterations are needed to find good heuristic functions, and for each iteration the system needs to

compile and run a whole suite of benchmarks. However, this optimization process need only be done once per installation of a compiler,[1] so there is little per-compilation overhead compared to compilers that use empirical feedback for every program that they compile.

The speedups achieved by a meta optimized compiler compared to a conventional optimizing compiler are non-trivial, but modest: typically in the low tens of percent [SAMO03]. An important caveat to consider with meta optimization is the risk of over-training. When a compiler is trained on a relatively homogenous suite of benchmarks, the resulting heuristic functions can actually be worse than the baseline conventional hand-tuned functions on programs outside the training set.

In summary, meta optimization appears to be a good method for optimizing complex heuristic functions. However, because the compiler is expected to work well on all or a large class of applications, it is unlikely that it will achieve the high performance of compilers that tune individual programs. Of course, the two techniques can be used together. In fact, meta-optimization can be applied not just to compilers, but to any program that uses complex heuristic functions.

5.2.3 Restructuring compilers with empirical feedback

A rich body of work on restructuring optimizations has grown over several decades (for example, [BGS94]). Applying a particular transformation at a particular program point is usually not hard. The major challenge faced by restructuring compilers is deciding where and with what parameters to apply restructuring optimizations. Conventional restructuring compilers use predictive models to estimate the impact of applying an optimization, but they do not validate these predictions by actually running the program. Because these models are inevitably not perfectly accurate, restructuring compilers tend to apply optimizations relatively conservatively in order to avoid having a large negative impact on the performance of the generated code.

Figure 5.3 illustrates the basic flow of restructuring compilers with empirical feedback. Empirical restructuring compilers offer enormous promise: in principle, they are capable of

[1]The optimization effort can be amortized across many identical (or sufficiently similar) systems, if a method for distributing the configured compiler is available.

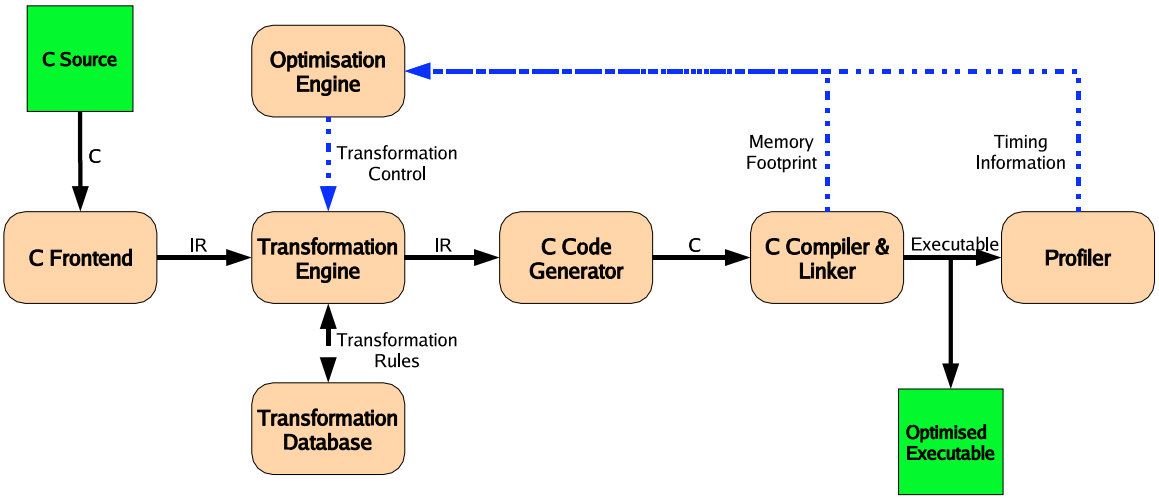


Figure 5.3: Empirical optimizing compiler flow. With minor variations, most program auto-tuning systems have a similar flow. Graphic borrowed from [FOTF05]. Automatic updating of the transformation database, as suggested by the bidirectional arrow suggests, is an uncommon feature of empirical compilers.

taking simple, unoptimized source code and automatically searching through the space of restructuring optimizations to find a high performance implementation. The major problem that empirical restructuring compilers have to contend with is long compile times. Typical restructuring compilers have many tens of (parameterized) optimizations that can be applied on their own or in combination at any of hundreds or thousands of program points. Clearly, the space of possible combinations of optimizations to try on a reasonably sized program is prohibitively large to search exhaustively.

In the last ten years, many efforts have been made to build an empirical restructuring compiler that uses a reasonable amount of compile time [FOK02, TVVA03, FCOT05, CGH+05, FOTF05, PE06, DCF+07]. The most impressive speedup results were reported in [FOTF05], and are reproduced in Figure 5.4. The authors used two independent strategies for pruning the enormous search space of possible optimizations. One strategy is completely random; it simply selects some arbitrary combination of optimizations at arbitrary program points and evaluates the quality of the result. The other strategy uses PBIL (Population-

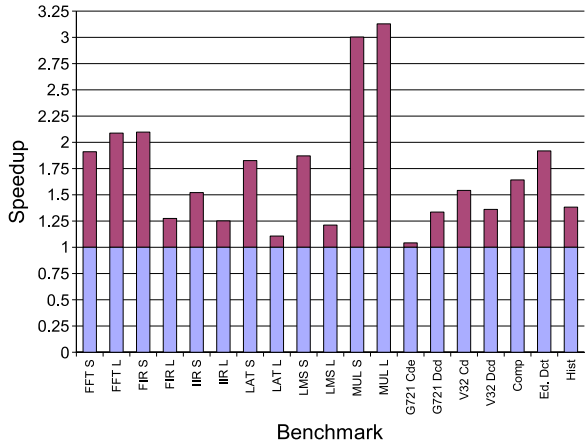


Figure 5.4: Aggressively optimizing compilers that use empirical feedback to tune particular programs can achieve impressive performance advantages compared to conventional compilers. The main disadvantage of this approach is very long compile times. Graphic borrowed from [FOTF05].

Based Incremental Learning) [Bal94], which combines elements of genetic algorithms and machine learning. The two search strategies have complementary strengths and weaknesses, though it is not clear from the paper whether the good optimizations found by the random search are used to inform the PBIL search.

The work in [FOTF05] has two important shortcomings. Even though the search strategies used explored only a small fraction of the total space of potential optimizations, compile times are still very long. Compilation of simple programs (in some cases, about 100 lines) can take hours. Also, the suite of benchmarks used includes programs that range from relatively simple to trivial. It is not clear how well this approach to compilation will scale to more complex programs.

Other compilers in this family tend to take more draconian approaches to pruning the search space. As a result, compile times tend to be shorter, but average performance improvement over conventional optimizing compilers is also more modest–typically in the low to mid tens of percent.

Aggressively applying empirical optimization strategies to a broad class of unoptimized

applications remains a fertile area of research, and it seems likely that some of the techniques used by these compilers will see wider usage. One particularly interesting avenue is pursued in [FCOT05]. The authors attempt to decrease the compile times needed for iterative compilation by, in some sense, blending empirical and model-based approaches. Using machine learning techniques and a relatively small number of actual compiled program runs, they dynamically build models that can predict the performance of untested configurations of a program. Their results are still preliminary, but this approach may lead to more practical iterative compilers. However, given the enormous search space that seems to be inherent in this approach there are still substantial challenges that these generic compilers must address in order to achieve the impressive speedups with reasonable compile times demonstrated by the more restricted auto-tuners covered in the next section.

5.3 The auto-tuner approach

Auto-tuners are not compilers in the usual sense, but either compilers that take a highly constrained DSL (Domain-Specific Language) as input or library generators that are designed to generate code with a predefined interface. Auto-tuners can use the extreme constraints on their inputs to restrict the space of implementations through which they search. Auto-tuners either generate or have programmed-in highly parameterized versions of algorithms. At installation time they search through the space of valid parameter values to find configurations that produce high quality results on a particular platform. In most cases, these parameters control the extent to which the loop transformations described in Section 5.1.2, and associated array transformations, are applied.

5.3.1 Linear algebra

Linear algebra operations, such as matrix multiplication, are central to a wide range of scientific and engineering algorithms. Two of the most popular linear algebra libraries are BLAS (Basic Linear Algebra Subprograms) and LAPACK (Linear Algebra PACKage).

ATLAS (Automatically Tuned Linear Algebra Software) [WPD01] is an implementation of BLAS and parts of LAPACK that is automatically tuned to a particular platform at installation time. Using analysis and extensive experience, the designers of ATLAS built

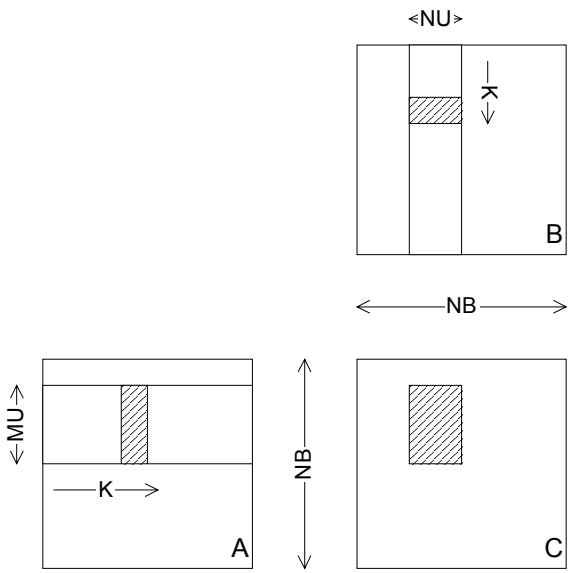
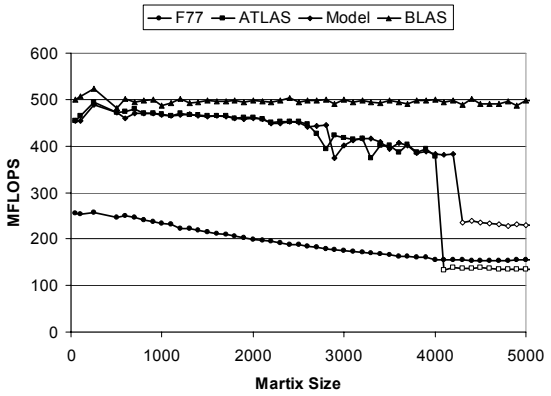


Figure 5.5: Blocked matrix multiplication with some of the parameters used by ATLAS to define a particular implementation. Graphic borrowed from [YLR+03].

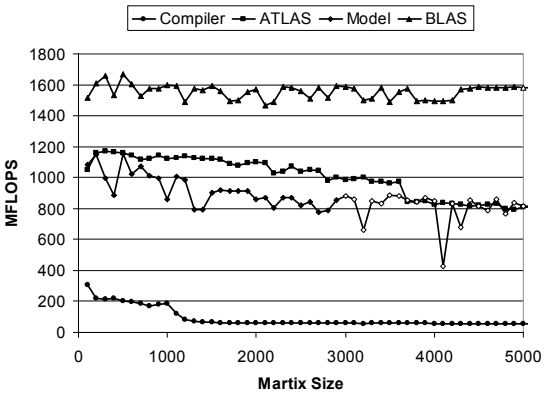
highly parametric versions of the BLAS and LAPACK routines, the most important of which is matrix-matrix multiplication (Figure 5.5). Most of the parameters determine the degree to which various loop transformations, like unrolling and blocking, will be applied. Some of the parameters relate to simpler architectural features, like whether fused multiply-adds are supported.

The original implementation of ATLAS uses a pseudo-global, "orthogonal line" empirical search. At installation time, the system runs a set of micro-benchmarks designed to discover properties of the system, like memory hierarchy structure. These properties are used to directly determine some of the simpler parameters. The more complex parameters are put in a priority order by the ATLAS developers. One by one, the values for the parameters are determined by compiling the library with every possible value of the parameter and choosing the value that leads to the best performance. Standard reference values are used for the parameters that have not yet been empirically determined.

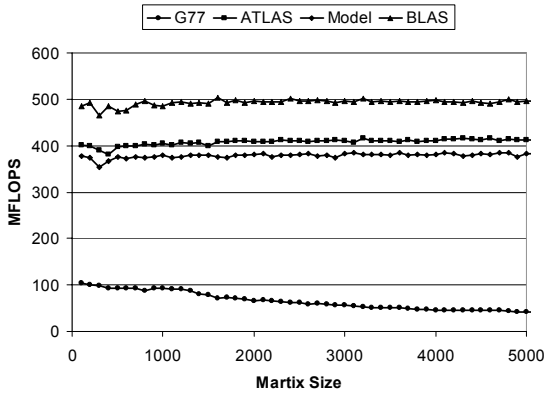
This method for choosing parameter values takes a relatively long time, but it is still



(a) SGI



(b) Sun



(c) Intel

Figure 5.6: Comparisons of empirically-guided optimization and model-based approaches to optimizing matrix-matrix multiplication. The three graphs are for three difference processor architectures. The main result is that empirical feedback provides a substantial advantage across a range of matrix sizes and architectures. The unfilled marks indicate a non-copying version of the algorithm. Graphics borrowed from [YLR+03].

linear in the number of parameters, compared to the exponential size of the whole configuration space. Typical installations take several hours. However, since installation needs to be performed only once per system, the installation time can be amortized over many uses of the library. Implicit in the one variable at a time search method is the assumption that the values of later-determined parameters do not have a significant impact on earlier-determined parameters. This assumption is not strictly true for any sufficiently complex target architecture, but it works reasonably well in practice. The search space must be pruned in some way; if ATLAS used a truly global search, installation times would be measured in months or years instead of hours.

Another approach to tuning ATLAS is combining test runs with models. [YLR+03, YPS05, CCH05]. In the work of Yotov, et al., Model-ATLAS uses the same micro-benchmarks as search-ATLAS, but then uses the architectural properties to estimate the best values for the parameters analytically. Some of the results of their initial study are presented in Figure 5.6. The three different graphs are for three different systems used, one each from SGI, Sun and Intel.

Before looking at the differences between model and search based ATLAS, consider two striking trends in this data. Hand-optimized libraries (represented by the BLAS line) achieve near-peak performance on all of the architectures across the range of matrix sizes. It is surprising that even for these relatively simple matrix computations that have arguably received more compiler optimization attention than any other application family, there is still a sizable performance gap between ATLAS and hand-optimized libraries. Perhaps even more striking is how poorly the code generated by an optimizing Fortran compiler performs. In many cases the compiler-produced code is more than an order of magnitude slower than the hand-optimized libraries.

The model-based version of ATLAS is able to generate code that is almost as high performance as the search-based version of ATLAS in most cases, and even higher performance in a few cases. Because extensive search is not necessary, model-ATLAS installation times are typically an order of magnitude or two shorter than search-ATLAS. In [YPS05], Yotov, et al. improved their model-based ATLAS by refining the model used to select the parameters, and using empirical search within a small neighborhood of parameter values around

- Choose largest N_B, which satisfies:

$$\left\lceil \frac{N_B^2}{B_{L1}} \right\rceil + 3 \left\lceil \frac{N_B}{B_{L1}} \right\rceil + 1 \leq \frac{C_{L1}}{B_{L1}}. \qquad (1)$$

- Choose M_U and N_U as follows:

$$N_U \leftarrow \left\lfloor \sqrt{N_R - L_s + 1} - 1 \right\rfloor$$
$$M_U \leftarrow \frac{N_R - L_s - N_U}{N_U + 1}.$$

- Use L_s as determined by the ATLAS micro-benchmark.
- Use FMA as determined by the ATLAS micro-benchmark.
- Set $F_F = 1$ and $I_F = N_F = 2$.

Figure 5.7: Equations from the basic model-based ATLAS for setting the parameters for dense matrix-matrix multiplication. It is remarkable how complex the equations are, given the relative simplicity of matrix multiplication as an application. The variables represent architectural features like cache size, cache line size, number of registers, and the presence of a fused multiply-add instruction. Graphic borrowed from [YPS05].

those chosen by the model. The hybrid model/search implementation of ATLAS performs better than both the simpler model and the pseudo-global search implementations. The improvements come from both the more sophisticated model and the local searching. Installation times for hybrid-ATLAS are still in the range of an order of magnitude shorter than search-ATLAS.

The models used by model- and hybrid-ATLAS are surprisingly complex, given the simplicity of the application (dense matrix-matrix multiplication). The equations for the simpler of the two models are shown in Figure 5.7. The equations for the refined model are roughly twice as complex. However, the combination of relatively short installation time and high performance that the hybrid-ATLAS implementation is able to achieve suggests that systems for tuning algorithms to particular architectures should use empirical search judiciously.

There are several other linear algebra-related auto-tuners that use optimization techniques similar to those described for ATLAS. FLAME (Formal Linear Algebra Method

Environment)[GGHvdG01] used an approach to dense linear algebra routines that recursively broke arrays into smaller blocks, instead of processing them primarily in row-major or column-major order. OSKI (Optimized Sparse Kernel Interface)[VDY05] is an extension of library tuning ideas to sparse matrix algorithms, which are substantially more complex and less regular than dense matrix algorithms. FEniCS (collection of Finite Element libraries)[LW10] is an autotuned library for differential equation solving, which is more complex in some ways than basic linear algebra routines.

5.3.2 DSP algorithms

Many DSP (Digital Signal Processing) transforms can be seen as a single linear algebra algorithm: the matrix-vector multiplication, with a constant matrix. This family includes the discrete cosine transform (DCT), discrete Fourier transform (DFT), Walsh-Hadamard transform (WHT), discrete Hartley transform (DHT) and discrete wavelet transform (DWT). SPIRAL[2][PMJ+05] is a system for generating efficient implementations of these kinds of transforms from descriptions written in SPL (Signal Processing Language).

SPIRAL exploits mathematical properties of the matrix of coefficients defined by a particular transform to find implementations that require significantly fewer operations than the $O(N^2)$ required for matrix-vector multiplication. For example, given a description of a DFT, SPIRAL can generate many different FFT (Fast Fourier Transform) implementations, which require only $O(N log(N))$ operations. In addition to these remarkable algorithmic transformations, SPIRAL is also capable of exploring the more pedestrian kinds of optimizations like loop unrolling and tiling that can have substantial constant-factor impact on the performance of a program.

How SPIRAL works

The high level architecture of SPIRAL is illustrated in Figure 5.8. It takes a linear transform, represented as an SPL formula, and applies two phases of optimization: “formula

[2]SPIRAL stands for either Signal Processing Implementation Research for Adaptable Libraries or Louis Auslander’s Remarkable Ideas for Signal Processing.

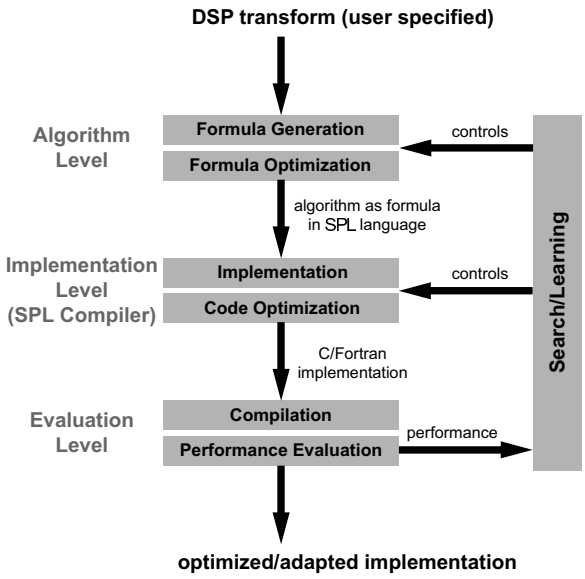


Figure 5.8: Architecture of SPIRAL. SPIRAL performs more radical program transformations than most auto-tuners, which it can do because the space of algorithms it accepts as input is quite narrow. Graphic borrowed from [PMJ+05].

optimization" and "code optimization". During the formula optimization phase, the formula is recursively broken down into several "smaller" SPL formulas. The smaller formulas can operate on smaller matrices or be simpler in some other sense. SPIRAL uses guarded rewrite rules to define the possible formula optimizations; for example, "if the coefficient matrix is upper-triangular, then the formula can be decomposed into three smaller multiplications and an addition." The guards for several rules may be satisfied by the same formula, which is where SPIRAL gets its flexibility in optimizing SPL formulas. Once a formula has been decomposed into sufficiently simple formulas, the formula optimization phase is declared complete, and the generated formula is passed to the code optimization phase.

The database of rules used by SPIRAL was written by hand, and is a distillation of about 50 journal papers on ways to implement linear transform algorithms. The space of possible decompositions of reasonably large transforms is enormous, so the search for good formulas must be guided in some way. Because the formulas can be defined recursively in

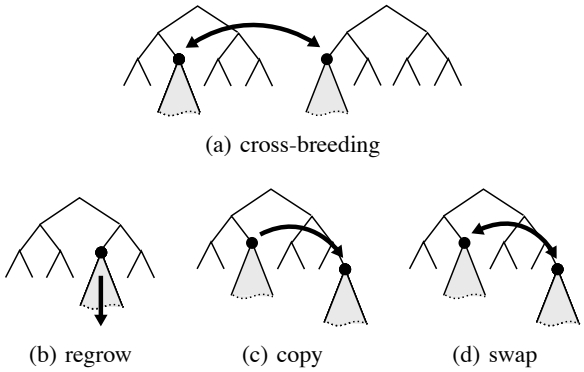


Figure 5.9: Moves used by the SPIRAL formula search algorithms. Splits in the trees indicate recursively splitting the input array, and potentially applying different algorithms to the subparts.

terms of smaller formulas (as suggested by Figure 5.9) dynamic programming works well as a search technique.

To understand how the dynamic programming search works, consider the following simple example. We want to implement formula A, which has two alternative decompositions: A_1 decomposes to sub-formulas B and C, and A_2 decomposes to D and E. Each of the sub-formulas has two alternative implementations, B_1, B_2, C_1, C_2, etc. SPIRAL will generate code for and test the running time of each of the sub-formulas to determine which is best; lets assume B_2, C_1, D_1 and E_2 are the winners. Now we implement A_1 and A_2 with their respective winning sub-formulas and test which is faster. Assuming A_2 is the winner, we now declare A_2, as implemented by D_1 and E_2, to be the best implementation of A. Notice that we did not even measure the performance of A_2 as implemented by D_1 and E_1, or any other combination. This search strategy is polynomial in the height of the formula tree, even though the space of all possible decompositions is exponential.

The SPIRAL group has also investigated an evolutionary approach to searching through the space of formula trees. This algorithm uses the "mutations" illustrated in Figure 5.9. Because the evolutionary search is more randomized than the dynamic programming search, it offers the promise of not getting "stuck" in local maxima. However, in practice, the

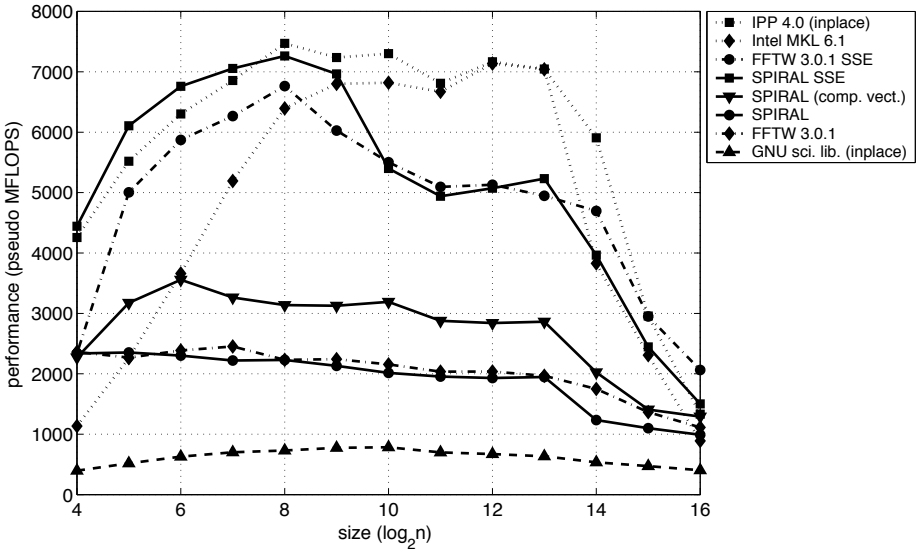


Figure 5.10: Performance on single-precision FFT [PMJ+05]

evolutionary search takes substantially longer than the dynamic programming search, and rarely produces superior formulas.

The optimizations performed in the "implementation" and "code optimization" phases of the SPL compiler are similar to the ATLAS optimizations. The SPIRAL optimization engine can iterate in the code optimization loop several times for a single formula produced by the formula optimization phase. The C or Fortran code produced by the code optimizer is then compiled with a conventional optimizing compiler.

A representative example of the performance of the code generated by SPIRAL is presented in Figure 5.10. This graph highlights several interesting performance trends. First, the code generated by both SPIRAL and FFTW [FJ05] is usually competitive with the hand-crafted libraries (IPP and Intel MKL). However, there is a substantial gap between the hand-crafted libraries and the generated code for FFTs of size roughly 2^{10} to 2^{13}. The SPIRAL designers speculate that this gap is caused by the generated code not being tuned to the size of the L2 cache well. This is a somewhat disappointing failure on the part of the auto-tuners, since in theory they should be able to adapt to any memory structure through empirical feedback.

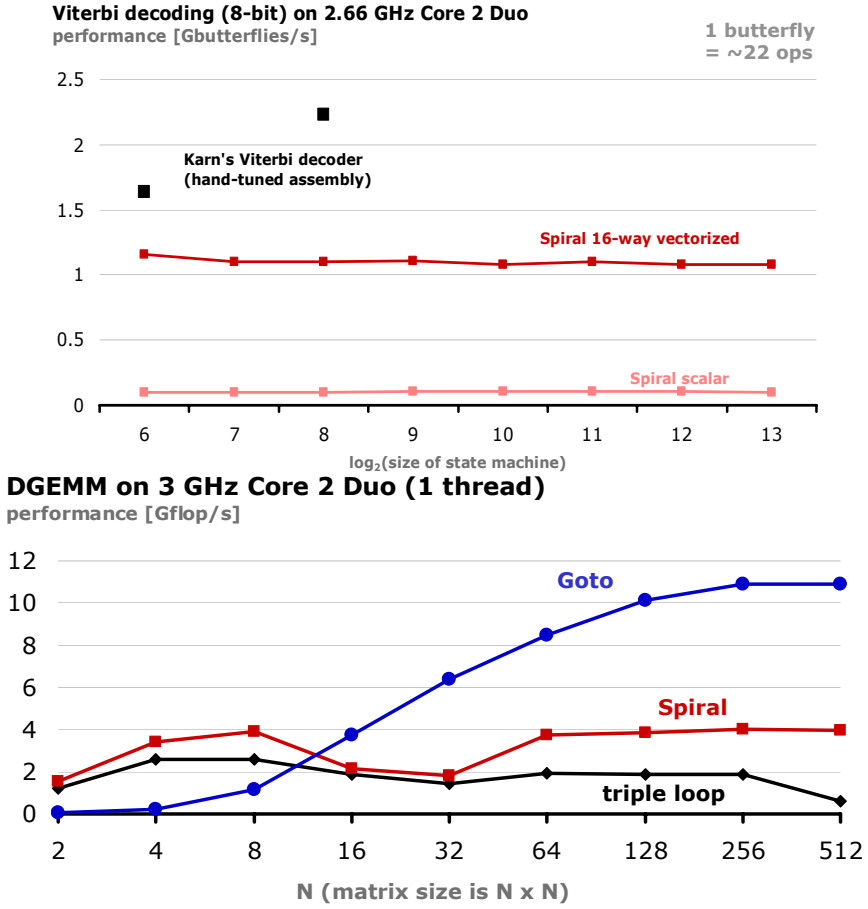


Figure 5.11: Non-matrix-vector multiplication algorithms in SPIRAL

Perhaps more strikingly, we see that the performance of a reasonably well-written FFT implementation from the GNU scientific library, compiled with a conventional optimizing compiler, is over an order of magnitude slower than the highest performance implementation for some input sizes. This evidence corroborates the ATLAS findings that given the complexity and unpredictability of today's processors, and even with only modestly complicated programs, there is a very large performance gap between reasonably well written code compiled with conventional optimizing compilers and highly optimized code—whether that optimization is done by expert programmers or automated systems.

The results achieved by the SPIRAL system are impressive, but there is an important caveat to keep in mind. The range of applications that SPIRAL can handle is quite narrow. Many useful DSP algorithms can be encoded as matrix-vector multiplication, but the space of programs that we would like to run on accelerators is much larger. In a recent presentation, the SPIRAL team described attempts to expand SPIRAL's repertoire to include Viterbi decoding and matrix-matrix multiplication. The preliminary results, shown in Figure 5.11 indicate that for some parameters, the SPIRAL-generated implementations are substantially slower than hand-crafted libraries.

The relative weakness of SPIRAL on applications outside of its original, relatively restricted domain raises doubts about the ability of highly optimizing auto-tuners to scale in generality towards general purpose programming languages. This work is still quite preliminary, though, so it is quite possible that more encouraging results will be forthcoming.

5.4 General purpose auto-tuning

Recently there has been an explosion in interest in applying auto-tuning to a wider range of applications [MSBL98, CH04, QKMC06, YSY+07, YW07, RPH+08, NBH+08, Sch09, TCC+09, KSP09, ZHCC09, HNS09, ZHCC09, GYQ10]. Developing an auto-tuner that can work for a wide range of applications is challenging because different applications have very different shapes (Figure 5.12). These images only begin to give a sense for the complexity that can exist in higher dimensional applications. Unfortunately, visualizing higher dimensional functions is challenging, which makes it hard to develop an intuition for such functions.

The most common search method used in these systems is some variant of *direct search* [KLT03], which is a broad and imprecisely defined class of optimization methods. In particular, methods based on the work of Nelder and Mead [NM65] are most common. The unifying concept behind these search methods (Figure 5.13) is that the next configuration to test is chosen by moving in some direction from the best configuration found so far. The specific methods for choosing the direction and distance usually use complex geometric heuristics. The other common property of direct search methods is that they do not use gradients to predict the location of the best configuration.

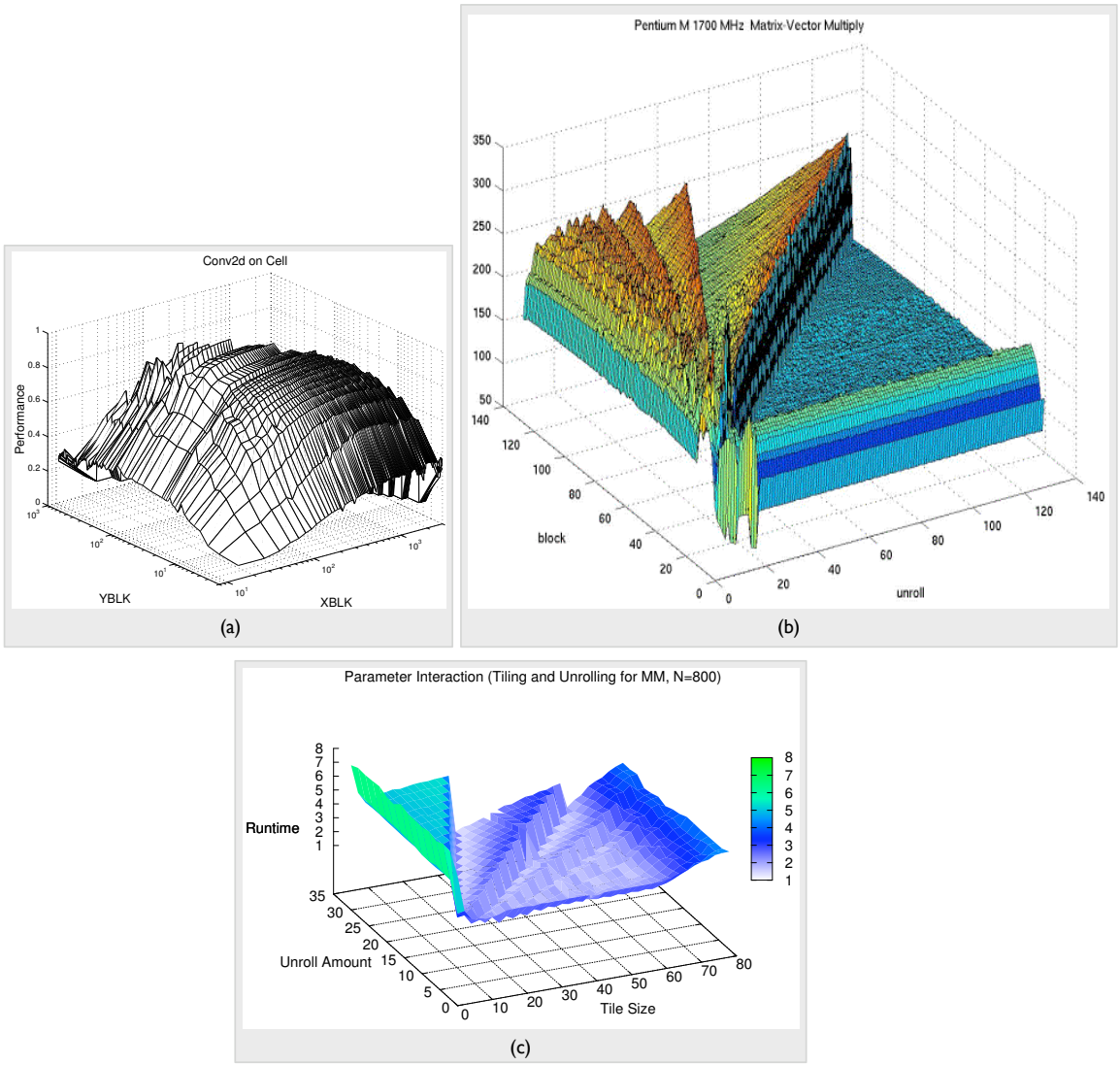


Figure 5.12: Illustrations of the complexity of the optimization functions faced by general purpose auto-tuners. These are three different applications from three different publications (a) 2D convolution [RPH+08], (b) Matrix-vector multiplication [YSD06], (c) Matrix-matrix multiplication [TCC+09].

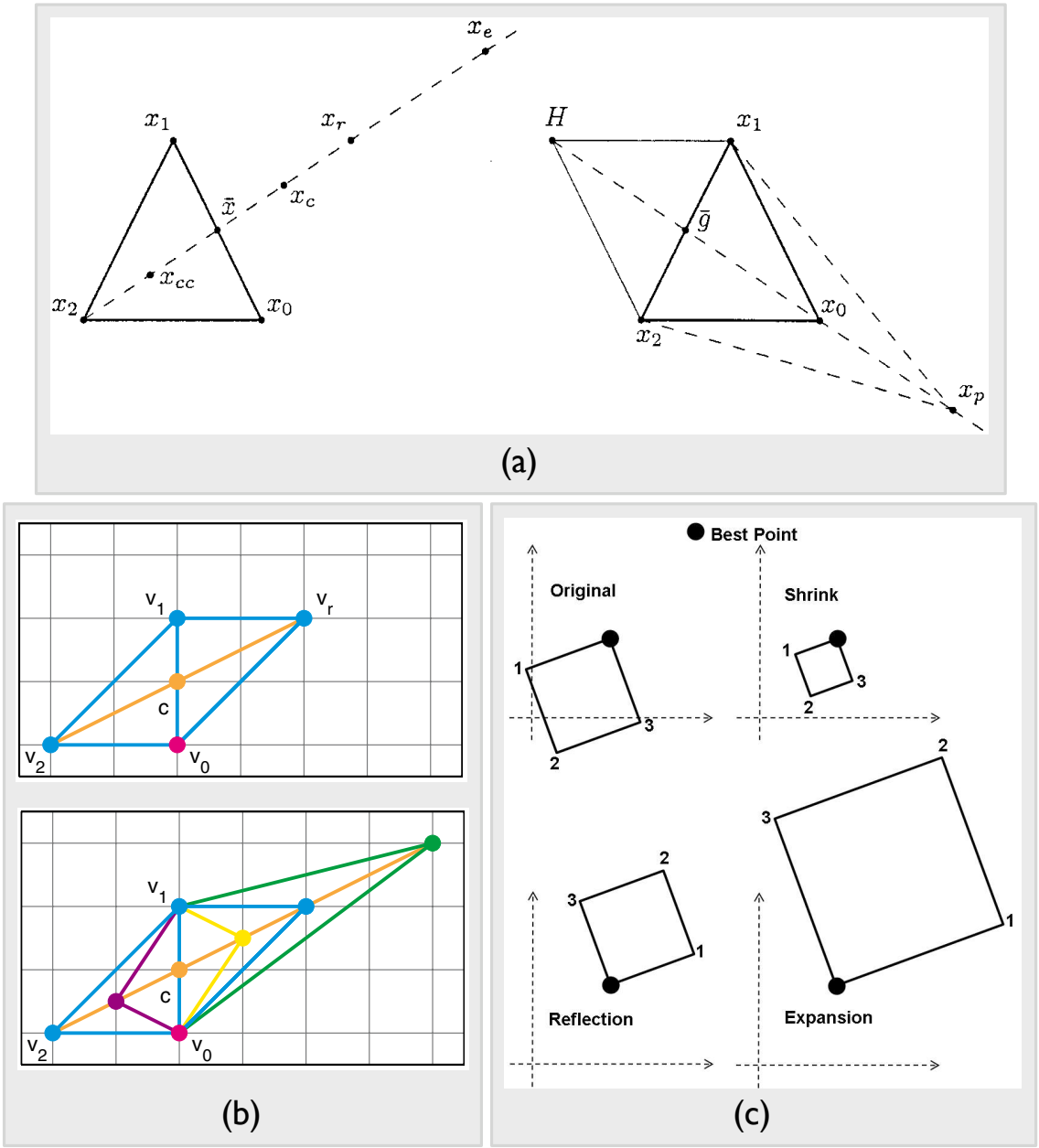


Figure 5.13: Direct search methods choose the next configuration to test by starting from the best found so far and moving in some direction based on the value of other configurations using geometrically complex patterns. Here are illustrations of the kinds of patterns used from three different publications (a) [PCB02], (b) [KLT03], (c) [TCC+09].

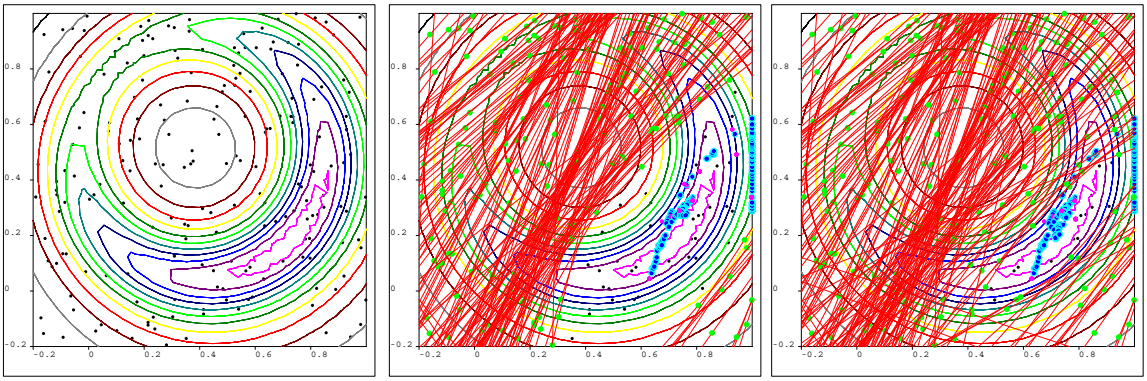


Figure 5.14: An example of an exotic search method from [MSBL98]. The concept is to iteratively reduce the size of the subspace that is believed to contain the best configuration by partitioning it with a single hyperplane. In each iteration a configuration is chosen to test in the "good" subspace, and another hyperplane is drawn. The "good" convex polyhedron is whittled down until it is small enough to test all configurations in it.

It seems that the most important reason for the popularity of direct search is that its mathematical properties (like convergence) have been studied extensively. However, most of the mathematical analyses start with the assumption that the function is smooth, which is clearly not always the case in real world tuning applications.

More exotic search methods for auto-tuning have also been proposed, such as Q2 [MSBL98] (Figure 5.14). The Q2 search method iteratively reduces the size of the subspace in which the best configuration is believed to exist. It is hard to compare the quality of these search methods, because every group doing auto-tuning work uses its own set of benchmarks. I believe that research on auto-tuning methods would benefit enormously from an effort to create an open, standard set of benchmarks.

In the absence of standard benchmarks and standard search methods, authors of new general purpose auto-tuning methods often resort to comparing against different versions of their own methods and very simple methods, like purely random searching. Some examples of these kinds of evaluations can be seen in Figure 5.15.

Auto-tuning is computationally expensive because the program being tuned must be

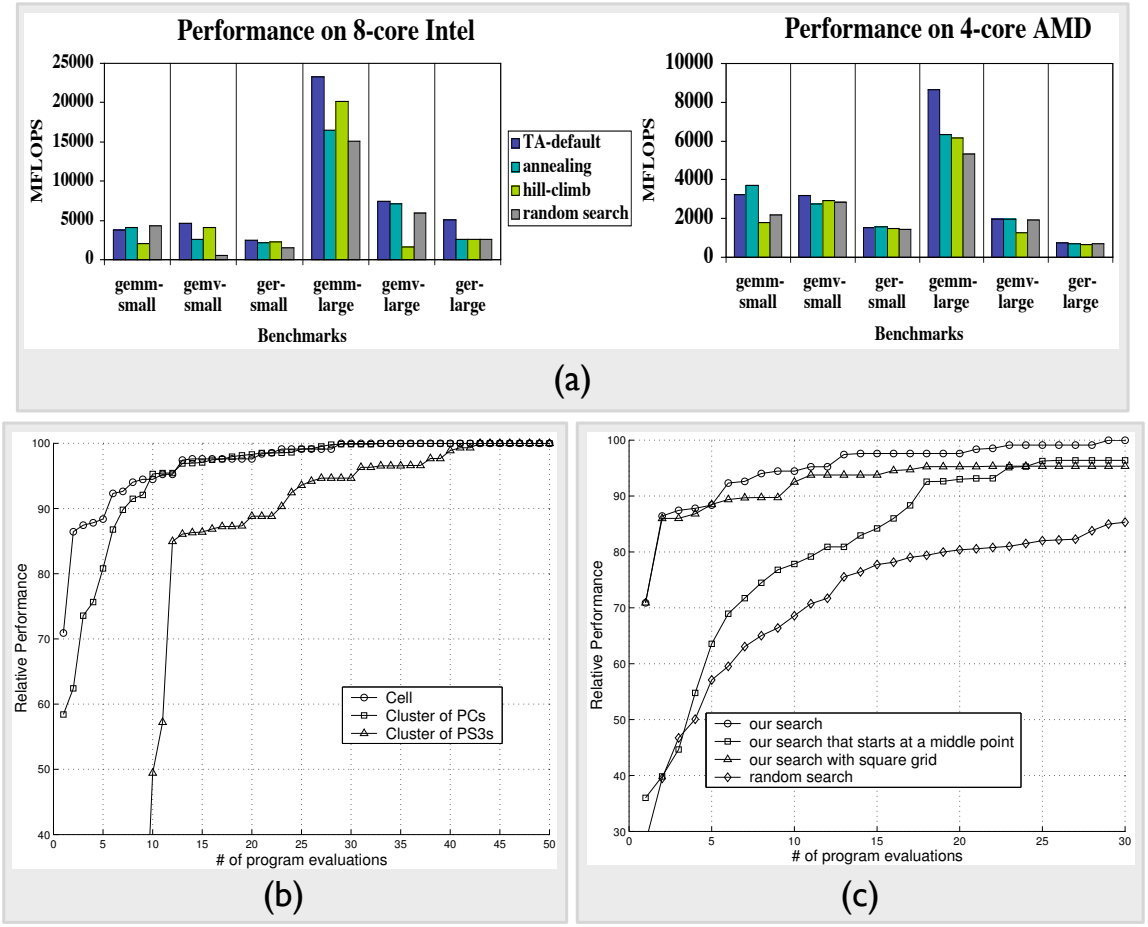


Figure 5.15: Evaluating tuning methods for general purpose applications is hard; it is not even clear what the best comparison metric is. (a) shows a comparison of the quality of the best configuration found by different methods [GYQ10]. (b) shows the quality of the best configuration found so far as a function of the number of tests [RPH+08]. This kind of data can be important for judging how quickly different search methods approach the optimal configuration. (a) and (c) show that researchers have not settled on a good baseline search method. Completely (pseudo-)random testing is still a common point of comparison.

compiled and run with some test input tens or hundreds of times. Parallelizing this process on a cluster of development machines could be an effective way to speed it up. However, many search methods do not parallelize nicely; deciding which configuration to test next depends on the results of testing all previously selected configurations. The method proposed in [TTH09] is in the direct search style, but tests multiple configurations that are geometrically related to the best found so far.

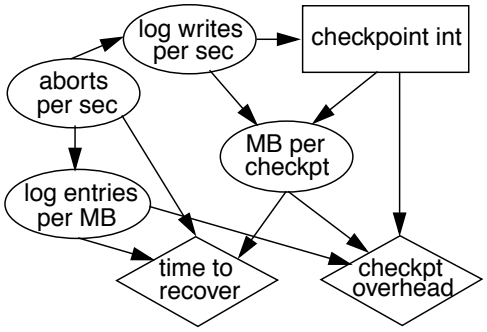
There has been some work on using auto-tuning methods on algorithms themselves, as opposed to tuning a program to an architecture [ACW+09, AWC+10]. This work is relevant to algorithms that can be tuned to have more or less accuracy or to converge more or less quickly in a way that is input data-dependent. Tuning is used to adapt the algorithm to a particular data set or class of data sets.

5.5 Tuning for coprocessor accelerators

Accelerators are substantially more complex than conventional processors in some ways. They have many semi-independent processing elements that communicate over a non-uniform network of some sort, many distributed on-chip memories, and nonstandard external memory interfaces. Moreover, these architectural complexities are exposed directly to the low-level programmer or compiler.

To achieve high efficiency on most reasonably complex kernels, on-chip memory and I/O resources must be used well. However, individual accelerators vary greatly in the size and structure of their on-chip memory and I/O systems. Therefore, a kernel must be adapted to the memory and I/O resources provided by a particular architecture. Auto-tuning is a promising approach to doing this adaptation.

The most important difference between accelerators and conventional processors for the purpose of auto-tuning is that accelerators have many hard constraints that must be met for an application to work at all. Such constraints make tuning harder because they make some configurations fail. Failing configurations have an undefined value for the application's quality function and most search algorithms have no natural mechanism to handle partially-defined quality functions. Moreover, many of the highest quality configurations "just barely fit", that is, they are very close to failing configurations in natural representations of the



P(MB per checkpt | 50 log writes/s, checkpt int)

MB per checkpt	checkpoint int 1 min	10 min	30 min	...
0-5	0.60	0.10	0.05	...
5-10	0.15	0.17	0.15	...
10-50	0.10	0.65	0.33	...
...	...	...	...	...

E(time to rec | log entr/MB, aborts/s MB/checkpt)

log entr/MB	aborts/s	MB/checkpt	time to rec
1000-5000	0-10	0-5	1.0 min
1000-5000	0-10	5-10	2.5 min
1000-5000	0-10	10-50	8.3 min
...	...	...	...

Figure 5.16: A different kind of tuning explored in [Sul03]. The goal is to discover the influence relationships between performance metrics of large complex applications, like relational database management systems and web servers. This is the only existing tuning system we are aware of that uses probabilistic or statistical methods in its tuning algorithms.

configuration space.

Another consideration for accelerators is that compiling an application for some architectures can take many hours. This is much more time than is expected in the conventional processor context, and it has a direct impact on the amount of tuning that is tolerable. This means that in order to be useful, auto-tuners for accelerators will need to find reasonably good configurations very quickly.

Existing approaches to auto-tuning with constraints assume that simple—in many cases linear—functions of the tuning parameters themselves can distinguish satisfactory configurations from unsatisfactory configurations [KLT03]. However, the relationship between application-level tuning parameters (degree of loop unrolling, buffer size, ...) and architecture-level resources (registers, instruction memory, ...) can be quite complex and hard to predict without actually evaluating a configuration. This means that constraints applied to application-level parameters either need to be quite conservative to avoid configurations that violate architecture-level constraints or the tuning system has to be designed to handle configurations with no defined quality.

The new search method we designed and evaluate in the next chapter uses a probabilistic

framework to combine the desire to find high quality configurations with the desire to avoid configurations that fail. We have found one other tuning system that uses a probabilistic approach [Sul03]. However, that system is designed for a very different kind of tuning (Figure 5.16). The systems being tuned are operating systems and network servers, and the goal of the tuning system is to discover, out of hundreds of factors, which actually influence each other. In a sense this work is focused on discovering a reasonable space for tuning out of the truly astronomical space of ways to tweak large, complex software systems. In performance tuning of kernels for accelerators, we generally assume that a reasonable configuration space is already defined.

Chapter 6

AUTO-TUNING FOR ACCELERATORS

All non-trivial applications for accelerators must be tuned or adapted to a particular architecture to achieve the best performance. Accelerator architectures have distributed resources like memories, registers, I/O ports and interconnect that need to be used efficiently. In contrast to conventional processors, accelerators have few features like global caches, branch predictors or reorder buffers that are designed to adapt to the needs of a program. Getting the best performance out of a conventional processor sometimes requires tuning; for accelerators it is almost always important. For programmability and portability it is critical that we hide the exact sizes of these resources from programmers.

The major difference between tuning for conventional processors and tuning for accelerators is that in accelerators there are many resources whose capacity is strictly limited. If an application attempts to use more embedded memory than exists in an FPGA or DSP, the application will simply fail. Failures are much less common for conventional processors, because resources like caches are generally designed with reasonably graceful dynamic fallback mechanisms.

As discussed in the previous chapter, there are many strategies for tuning applications to particular architectures. In this chapter I describe a semi-automatic empirical tuning method where the programmer specifies the variables to tune (or tuning knobs) and application-level optimization criteria. The compiler writer or architect supplies system-specific resource constraint formulas. The tuning system automatically searches for configurations that satisfy all constraints and have high quality according to the optimization formula. Below I discuss the strengths of this approach compared to other tuning strategies, in the context of accelerators.

Failures make auto-tuning more challenging because it is no longer sufficient to optimize a single quality function. It is *possible* to define the quality of all failing configurations to

be "very low". However, there are two important weaknesses to this simple approach to failures:

- If a large portion of the configurations fail, it can be hard for a tuning search to find any regions of the space where there are successes, because all failures are viewed as equally bad.
- The highest quality configurations are often very close to failing configurations, because it is usually best to use up all the available resources without oversubscribing them. So it is likely that smart auto-tuning algorithms for accelerators will spend a lot of time exploring the border between successful and failing configurations. Understanding *why* some configurations fail can help the search choose better sequences of configurations to test.

In this chapter I describe an auto-tuning system with two novel features:

- It is designed to simultaneously optimize a quality formula and avoid testing too many failing configurations. Previous work implicitly assumes that all configurations have a defined quality value. Our key technical innovation is that predictions about untested configurations are made in terms of probabilities and probabilistic value distributions; this allows multiple factors to be combined in a mathematically clean way.
- The system is integrated into a programming language. This is important in the context of accelerators because the need for tuning is so common. Concurrently with the development of tuning knobs in Macah a few similar proposals have been published [YSY+07, ACW+09, HNS09, SPT09, ZHCC09, AWC+10].

The probabilistic framework for tuning has some additional benefits that are not necessarily limited to accelerators. We will discuss these throughout the chapter.

6.0.1 Portability

Tuning applications to a single accelerator is a challenging problem, and providing effective automation for it is an important usability goal for accelerators. Solving the tuning

problem well also contributes to the goal of *performance portability* across different accelerator architectures. Portability is an extremely useful property of programming systems and individual programs. The ability to develop a program on one platform, then rebuild it on another with little or no additional engineering effort, saves time and money. Portability also contributes to freeing system designers and application developers to innovate independently.

Portability across different kinds of accelerators (for example GPU to FPGA) is something that does not exist at all in current practice. Even porting between different vendors of similar accelerators (for example, Altera to Xilinx or NVIDIA to ATI) generally requires substantial manual effort. Porting to a later generation of a particular product is sometimes supported, but often does not bring any performance benefit because the application does not automatically scale up.

The reason for this portability gap is that engineers who choose to use an accelerator are usually concerned with getting close to the best performance possible for their application, and getting peak performance out of an accelerator requires fitting the application to particular architectural capacities, like the embedded memory structure. An application that is manually tuned to a particular architecture quite possibly will not work at all on a different architecture, and almost certainly will not achieve peak performance.

Improving portability is one of the ways in which this dissertation contributes to bringing accelerator programming to the C-level. One of the important differences between C and less abstract languages is that a program developed in C on machine X should be a simple recompile away from running reasonably well on machine Y.[1] Not only do C programs work correctly when recompiled on different machines, they also often perform well even if the architectures in question have very different performance characteristics. This kind of performance portability is an important part of the meaning of "C-level".

The tuning system described in this chapter is a direct solution for the problem of porting between accelerators of different capacities in a particular family. Porting between different vendors or kinds of accelerators is a harder problem because in some cases there

[1] As long as the program is written in a portable style.

are kinds of resources present in one family that are totally absent in another. A simple example is floating-point arithmetic which is extremely costly to implement on architectures that do not support it natively. It is possible that the tuning ideas presented here can be extended to cover these more challenging porting scenarios, but we have not investigated that question deeply.

6.1 Overview of the tuning knobs method

There are four basic ingredients required to use the tuning knob system:

- A program with tuning knobs, as well as code to indicate what program features (like run time) the system should record.
- A testing harness supplied by the programmer that the system will compile and run in particular configurations.
- A real-valued optimization formula written by the programmer, with program features as the variables and simple arithmetic like addition and multiplication. Also, the programmer must indicate whether the system should search for high or low values of the formula.
- A set of Boolean-valued constraint formulas, some of which are written by the programmer (e.g., energy consumed less than some application-defined limit) and some of which are provided as part of the system implementation (e.g., memory usage less than capacity of the target architecture).

From the programmer's perspective, a tuning knob is a special kind of "constant" that is defined to have a range of possible values instead of a particular value. Tuning knobs can be used anywhere a constant is used, including as the size of an array.

The tuning process involves iteratively selecting and testing configurations until some stopping criterion is met. The search algorithm has to make predictions about which untested point is most likely to both have a good value for the optimization formula and satisfy all the constraint formulas. One of the contributions of my work on tuning knobs is the idea that probabilities and probabilistic distributions should be used to represent predictions about the values of program features, the likelihood of meeting constraints, and

the likelihood of having a "good" value for the optimization formula. Casting the problem in probabilistic terms is useful because we can use rich statistical math to combine many competing factors.

Aside. The tuning system currently supports only tuning knobs that can take any `int` value between a specified minimum and maximum. There are other kinds of knobs that would be interesting extensions:

- `float`-valued (or `double`-valued) knobs are a straightforward extension that could be implemented with very little change to the system.
- Knobs with an unordered set of possible values, which we call *modal knobs* or *switches*, present a greater challenge to the system because it is much harder to say what the relationship is between configurations that differ in the value assigned to a switch. Switches can be simulated with a basic `int` knob by encoding each value as an integer, but the value predictions during the search process might be very inaccurate. Supporting switches is an interesting direction for future work.
- Constrained versions of any kind of knob can be supported using a constraint formula. For example, consider an application with two knobs, N and M, where it does not make sense for N to be greater than M. The knobs can be specified in the normal way, and the programmer can provide an extra constraint formula, $(N \leq M)$. The system will learn to not test configurations that violate the constraint.[2]

6.2 An example

To show how the Mosaic group uses tuning knobs in the applications we have developed, we go through an iterative refinement of a finite impulse response (FIR) filter. We start with a simple sequential version and then add in refinements that make the code more accelerator-friendly.

Figure 6.1 shows a simple sequential FIR filter in Macah (or C). Each entry in the output array (`O`) is defined to be the scaled sum of a window of the input array (`I`), with the

[2]Our tuning search algorithms are designed to handle the more challenging case of constraints that are not simple combinations of the tuning parameters. It would be easy to add a special case for slightly more efficient handling of the simpler constraints.

```
1 void fir1(int *I, int *O, int *C, int N, int NC)
2 {
3   for (j = 0; j < N; j++) {
4     O[j] = 0;
5     for (k = 0; k < NC; k++) {
6       O[j] += I[j+k] * C[k];
7 } } }
```

Figure 6.1: A simple sequential FIR filter.

```
1 void fir2(int *I, int *O, int *C, int N, int NC)
2 {
3   for (j = 0; j < N; j++) {
4     O[j] = 0;
5     FOR (k = 0; k < NC; k++) {
6       O[j] += I[j+k] * C[k];
7 } } }
```

Figure 6.2: FIR with an unrolled inner loop.

scaling coefficients given in `C`. The number of coefficients is `NC`, and the number of values to compute is `N`. In order to avoid running off the end of the input array, we assume that `I` is larger than `O`.

There is abundant potential parallelism in this application. All $\texttt{N} \times \texttt{NC}$ multiplications are completely independent, and the $\texttt{N} \times \texttt{NC}$ additions can be broken up into `N` independent reductions of size `NC`. In Macah we do not assume the system will automatically determine how to exploit this parallelism, so the programmer has to provide more implementation detail.

The first step in parallelizing this program for accelerators, illustrated in Figure 6.2, is to request that the inner loop be completely unrolled. As long as the number of coefficients is reasonably small, this is not a bad strategy. However, it is important to think about where the data is going to be stored. In particular, the accesses to global data structures inside the inner loop are problematic.

The next refinement, shown in Figure 6.3, is to declare extra buffer arrays to avoid global data structure access in the inner loop. This refinement creates the awkward problem that we want to declare buffers whose size is the number of coefficients, but the original code had the number of coefficients as a parameter to the FIR function. We need the

```
 1 #define NC ...
 2
 3 void fir3(int *I, int *O, int *C, int N)
 4 {
 5   int Ib[^NC^], Cb[NC];
 6   for (int j=0; j<NC; j++) {
 7     Ib[j] = I[j];
 8     Cb[j] = C[j];
 9   }
10   for (j = 0; j < N; j++) {
11     int Ob = 0;
12     FOR (k = 0; k < NC; k++)
13       Ob += Ib[k] * Cb[k];
14     O[j] = Ob;
15     Ib <<= 1;
16     Ib[NC-1] = I[j+NC];
17 } }
```

Figure 6.3: FIR with explicit local buffering.

number of coefficients to be defined at compile time to know if the buffer arrays can fit in the workspace memory of the target architecture. It would be cleaner to use a meta-programming mechanism like C++ templates,[3] but to keep the focus on the main topic of tuning knobs, we assume that `NC` is a globally defined constant.

The input buffer `Ib` is a shiftable array,[4] which is a perfect fit for the FIR filter, because all but one of the values from the input array that are used to compute output i are used again to compute output $i + 1$. With these new buffer variables, we have some extra bookkeeping code for initializing them (lines 6-9), moving data from and to the global data structures (lines 14 and 16), and managing the movement of data within the workspace memory (line 15). The important feature of this version 3 is that there are no longer any accesses to the global data structures in the inner compute loop.

Next, we consider the case where the number of coefficients is quite large (say, tens of thousands). Version 3 has a problem in this case, because the inner loop is unrolled as many times as there are coefficients. If the number of coefficients is large, this will create a program with an unreasonably huge amount of code. The solution to this problem, presented in Figure 6.4 is to break the buffers up into banks and tile the inner loop into two

[3] . . . or something cleaner than C++ templates.

[4] See Section 3.3.6 for the definition of shiftable arrays.

```
 1  #TuningKnob Banks int 1..1024
 2  #define NC ...
 3  #define CperB ((NC-1)/Banks)+1
 4  #define NC_round Banks * CperB
 5
 6  void fir4(int *I, int *O, int *C, int N)
 7  {
 8    int Ib[Banks][^NC/Banks^];
 9    int Ob[Banks];
10    int Cb[Banks][NC/Banks];
11    for (j=0; j<Banks; j++) {
12      for (k=0; k<CperB; k++) {
13        int n = j * CperB + k;
14        Cb[j][k] = n < NC ? C[n] : 0;
15        Ib[j][k] = I[n];
16      }
17    }
18    for (j = 0; j < N; j++) {
19      FOR (b = 0; b < Banks; b++)
20        Ob[b] = 0;
21      for (k = 0; k < NC/Banks; k++) {
22        FOR (b = 0; b < Banks; b++) {
23          Ob[b] += Ib[b][k] * Cb[b][k];
24      } }
25      int o = 0;
26      FOR (b = 0; b < Banks; b++) {
27        Ib[b] <<= 1;
28        Ib[b][CperB-1] = IB[b+1][0];
29        o += Ob[b];
30      }
31      O[j] = o;
32      Ib[Banks-1][CperB-1] = I[j+Banks*CperB];
33  } }
```

Figure 6.4: Banking the buffers.

nested loops, one of which will be unrolled and one of which will not.

The input and coefficient buffers, declared on lines 9 and 11 respectively, are now two dimensional arrays, and the assumption is that they will be partially scalarized[5] into several smaller arrays, each of which can be accessed independently. We assume that the banks of the buffer arrays can be allocated into the distributed local memories in the target architecture. If the number of coefficients was so large that the buffers could not fit, additional programming effort would be required to add an extra layer of shuffling data back and forth between the local and global data structures.

The idea of banking an array for parallel access is a common one, and it always brings with it the question of how many banks to use. This is where the tuning knob comes in.

[5]See Section 3.5.6 for a discussion of array scalarization.

```
 1  #TuningKnob Banks int 1..1024
 2  #TuningKnob AccPerB int 1..32
 3  #define NC ...
 4  #define CperB ((NC-1)/Banks)+1
 5  #define NC_round Banks * CperB
 6
 7  void fir5(int *I, int *O, int *C, int N)
 8  {
 9    int Ib[Banks][^NC/Banks^];
10    int Ob[Banks];
11    int Cb[Banks][NC/Banks];
12    for (j=0; j<Banks; j++) {
13      for (k=0; k<CperB; k++) {
14        int n = j * CperB + k;
15        Cb[j][k] = n < NC ? C[n] : 0;
16        Ib[j][k] = I[n];
17      }
18    }
19    for (j = 0; j < N; j++) {
20      FOR (b = 0; b < Banks; b++)
21        Ob[b] = 0;
22      for (k = 0; k < NC/(Banks*AccPerB); k++) {
23        FOR (b = 0; b < Banks; b++) {
24          FOR (k2=0; k2<AccPerB; k2++) {
25            Ob[b] += Ib[b][k+k2] * Cb[b][k+k2];
26      } } }
27      int o = 0;
28      FOR (b = 0; b < Banks; b++) {
29        Ib[b] <<= 1;
30        Ib[b][CperB-1] = IB[b+1][0];
31        o += Ob[b];
32      }
33      O[j] = o;
34      Ib[Banks-1][CperB-1] = I[j+Banks*CperB];
35  } }
```

Figure 6.5: Multiple parallel accesses to each bank.

As you can see on line 1, the number of banks is not fixed in the program, but allowed to range between 1 and 1024. On most accelerators this program will perform best when the number of banks is roughly equal to the number of distributed memories in the architecture. Instead of fixing that number into the program source itself, tuning knobs let us leave that decision to the compiler.

This tuning knob also raises the critical issue of failures. On some systems it might not be possible to allocate multiple arrays into a single memory, so if the number of banks is greater than the number of memories available the program will not compile.

The final refinement addresses the issue of multiple parallel accesses to a single bank. The code in Figure 6.5 shows one approach. This transformation is useful for reuse of distributed

embedded memories when the implementation does not support packing of multiple arrays into a single hardware memory. This optimization is more implementation-specific than the others we looked at, and is a good concrete example of the level of portability that we have achieved with Macah. Tuning knobs and other language features make it relatively easy to port Macah programs between architectures, as long as those architectures are within the same family. There are more thoughts on the ease/difficulty of porting between different kinds of accelerators in the conclusions chapter.

It is important to keep in mind that this kind of application refinement in Macah is not a perfectly architecture-independent process. There are some assumptions in this code about the structure of the memory system that may be more or less appropriate for different accelerators. As mentioned in the introduction to this chapter, tuning knobs make porting between different sizes of the same kind of architecture easy, but porting between different kinds of architecture requires at least some careful forethought by the programmer, and perhaps a modest amount of additional engineering effort for the port.

6.3 Context for accelerators

Given the potentially huge space of legal knob settings for a program, and the relatively long time that compilation and testing for accelerators can take, it is important for a knob-setting search procedure to be able to find good settings with only a very sparse sampling of the whole space. To be concrete, the challenges are:

- Resource limits, like workspace memory size, interconnect resources, or instruction/ configuration storage, can result in certain settings failing to produce a score.
- Evaluating a single point can take minutes to many hours.
- Complex heuristics in toolchain and unrelated runtime system events can add a non-trivial amount of noise to the quality measurements.
- Plateaus, which make it hard to decide on a search direction, are common because for some knobs all values within certain sub-ranges are essentially equivalent.
- Local minima and maxima are common, and are a challenge for many optimization strategies.

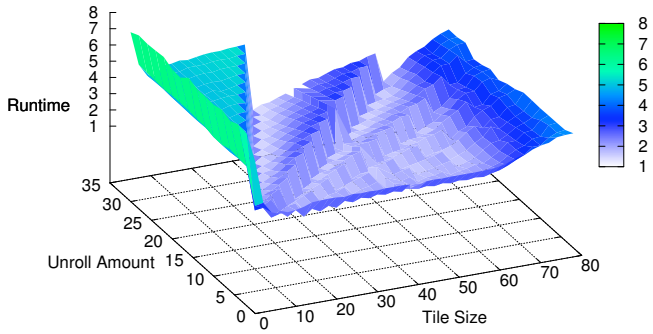


Figure 6.6: A tuning space for a matrix multiplication kernel on some architecture. The run time function is not generally smooth, which makes tuning a challenge. In particular, there are flat "plateaus", sharp "cliffs" and multiple "troughs" with near-optimal values. (Graphic borrowed from [TCC+09].)

Figure 6.6 shows an example of the kinds of challenging shapes that are common in auto-tuning.

To perform well, the search must balance three competing concerns:

1. The search should favor areas where there are likely to be high-value points.
2. The search needs to "intelligently" avoid over-testing in regions where the compilation process or application is likely to fail.
3. The search should not completely avoid searching in areas of lower predicted value, because hard to predict interactions between compiler, architecture and application features can cause nonlinearities in the value function.

In other words, the desirability of some untested point is a function of three things: its expected goodness value, the probability of it succeeding, and the confidence level of these predictions.

6.4 The prominent alternatives

As described in the previous chapter, there are several methods for tuning applications to particular target architectures:

- Mostly manual human effort
- Mostly automatic optimization by an aggressive transforming compiler
- Using domain knowledge to calculate parameter settings from an architecture/system description of some kind
- Empirical auto-tuning

All of these methods are appropriate under certain circumstances. The strengths of empirical auto-tuning make it a good choice for compiling C-like languages to accelerators. We will consider some of the important weaknesses of the other methods for this task.

6.4.1 Mostly manual human effort

It is clearly possible for a programmer to tune an application to a particular architecture by hand. Manual programmer effort has the obvious cost of requiring lots of human time, which can be quite expensive. In particular, when programs have to be retuned by a human for each architecture, it is not possible just to recompile an application when a new architecture is released. So if portability is a concern, fully manual tuning is generally not the best choice.

6.4.2 Mostly automatic compiler optimization

Fully automatic optimization has the extremely desirable feature of not requiring any extra tuning effort from the programmer. Purely static compiler optimization faces the daunting challenge of not only adjusting the size of various loop bounds and buffers, but deciding what higher-level transformations should be applied as well. There is little to no opportunity for human guidance in this situation. The space of all possible transformations of even a simple program can be intractably large, and decades of research on auto-parallelization so far has not led to compilers that can reliably navigate this space well. In fact, as argued by the authors of [CDG+06], the gap between vanilla code compiled by the best optimizing compilers and hand-optimized code has grown over the years as architectures have become more complex.

6.4.3 Using deterministic models

Deriving application-level parameters from architecture-level parameters with formulas like "use half of the available memory for buffer X" are sometimes a good tradeoff between human effort and application performance. However, the more complex the application and architecture in question, the more complex these relationships are. Even the interactions between two levels in a memory hierarchy can be complex enough to produce relationships that are extremely hard to capture with formal models.

Another approach that fits in this category and is reasonably common in the FPGA space is writing a "generator" (usually in some scripting language) that takes a few parameters and produces an implementation tuned to the given parameters.

6.4.4 Empirical tuning

The space of empirical auto-tuners includes: (1) self-tuning libraries like ATLAS[WPD01], PhiPAC[BACD97], OSKI[VDY05], FTTW[FJ05] and SPIRAL[PMJ+05]; (2) compiler-based auto-tuners that automatically extract tuning parameters from a source program; and (3) application-level auto-tuners that rely on the programmer to identify interesting parameters. The tuning knob search developed in this chapter fits primarily into category 3, though our search methods could certainly be used in either of the other contexts.

6.5 How it works

In this section, we describe the most important features of the tuning knobs system in a top-down style. There are many implementation details in the system, and many of them have a number of reasonable alternative choices. We will explicitly point out what parts are essential to the functioning of the system and what parts could be replaced without fundamentally altering it.

In theory, at each step in the search process the algorithm is trying to find the untested configuration c that maximizes the following joint probability formula:

$$P(c \text{ is the highest quality}^6 \text{ configuration} \cap c \text{ satisfies all constraints})$$

Analyzing the interdependence of the quality and constraint features of a tuning space is hard given the relatively small amount of training data that auto-tuners typically have to work with. There certainly are such interdependences, but we found in our experimentation that it works reasonably well to assume that the two factors are completely independent. With this independence assumption we can factor the joint probability into the product of two simpler probabilities.

$$P(c \text{ is the highest quality configuration}) \times P(c \text{ satisfies all constraints})$$

A successful tuning algorithm must maintain a balance between focusing the search in the areas that seem most promising versus testing a more evenly distributed set of configurations to get a good sampling of the space. There is a nice technical trick that helps maintain this balance, which is that instead of predicting the probability that a candidate is the very highest quality, we predict the probability that the quality of a candidate is higher than some target; how the target is adjusted is explained in Section 6.9.1.

$$P(\text{quality}(c) > q_t) \times P(c \text{ satisfies all constraints})$$

Next we consider how to compute the joint probability that a configuration satisfies all constraints. Ideally, the system would be able to model the correlation between different constraints and use them to predict the joint probability of satisfying all constraints. Unfortunately, the small number of configurations that auto-tuning systems generally test provide very little training data for these kinds of sophisticated statistical analyses. However, there are often strong correlations between different constraints, since most of them relate to overuse of some resource, and a knob that correlates with one kind of resource consumption often correlates with consumption of other kinds of resources. In our experiments we found that using the minimum probability of success across all constraints worked well. This is an optimistic simplification; if we assume instead that all constraints are completely

[6]For simplicity of presentation we assume that the optimization formula specifies that high values are good.

independent, using the product of the individual probabilities would be appropriate.

$$P(\text{quality}(c) > q_t) \times \min_{N \in \text{constraints}} \big(P(c \text{ satisfies constraint } N)\big)$$

At each step in the tuning process, the system attempts to find the untested configuration that maximizes this formula. This formula is complex enough that it is not clear that there is an efficient way to solve precisely for the maximum. Instead our tuning algorithm chooses a pseudorandom set of candidates, evaluates the formula on each one, and tests the one with the highest probability.

To evaluate this formula, we need probabilistic predictions for the program features that determine quality and constraint satisfaction. We call raw features, like the run time or energy consumption of the program *sensors*. Sensors can be combined with arithmetic operations to make *derived features*, like run time-energy product.

For each candidate point and each sensor, the tuning knob search algorithm produces a predicted value distribution for that sensor at the given point. We use normal (Gaussian) distributions, because as long as we assume that the features are independent, we can combine normal distributions with arithmetic operations to produce predictions for the derived features that are also normal. Derived features are discussed in more detail in Section 6.7.

Aside. In our experiments, we made the simplifying assumption that a particular configuration will always have the same values for its features (run time, memory use, ...) if it is compiled and tested multiple times. In other words, we assume that the features are a deterministic function of the tuning parameters. This is clearly not true in all systems, and accommodating random system variation is an interesting and important direction for future work. It seems possible that probabilistic predictions, as implemented in the tuning knob search, will be useful in the context of the random variation problem as well. Since we already represent predictions for untested points as distributions, representing values for tested points as distributions may not be a big challenge.

6.5.1 Hard to predict constraints

One of the trickiest issues left to deal with is deciding what the constraint formula should be for some failure modes. The easy failures are those for which some program feature can be used to predict failure or success, and it is possible to get a value for that feature for both failed and successful tests. For example, the programmer can impose the constraint that the program cannot use more than a certain amount of energy. Every test can report its energy use, and these reported values can be used to predict the energy use of untested configurations.

The harder failures are those for which the most logical feature for predicting the failure does not have a defined value at all for failing tests. For example, consider an application where some tuning knobs have an impact on dynamic memory allocation, and a non-trivial range of configurations require more memory than is available in the system. It is possible to record the amount of memory used as a program feature, but for those configurations that run out of memory it is not easy to get the value we really want, which is *how much memory would this configuration have used if it had succeeded.*

Another constraint of the nastier variety is compile time. Full compilation for FPGAs and other accelerators can take hours or even days, and it can be especially high when the resource requirements of the application are very close to the limits of the architecture. For this reason it is common to impose a time limit on compilation. Like the memory usage example, failing configurations do not tell us how much of the relevant resource (compile time) would have been required to make the application work.

To compute predictions for the probability of satisfying the harder constraints, we use proxy constraints that the programmer or compiler writer believes correlate reasonably well with the "real" constraint, but for which it is possible to measure a value in both successful and failed cases. An example of a proxy constraint from our experiments is the size of the intermediate representation (IR) of a kernel as a proxy for hitting the time limit during compilation. This is not a perfect proxy in the sense that some configurations that succeed will be larger than some that cause a time limit failure. This imperfection raises the question of what the IR size limit should be for predicting a time limit failure.

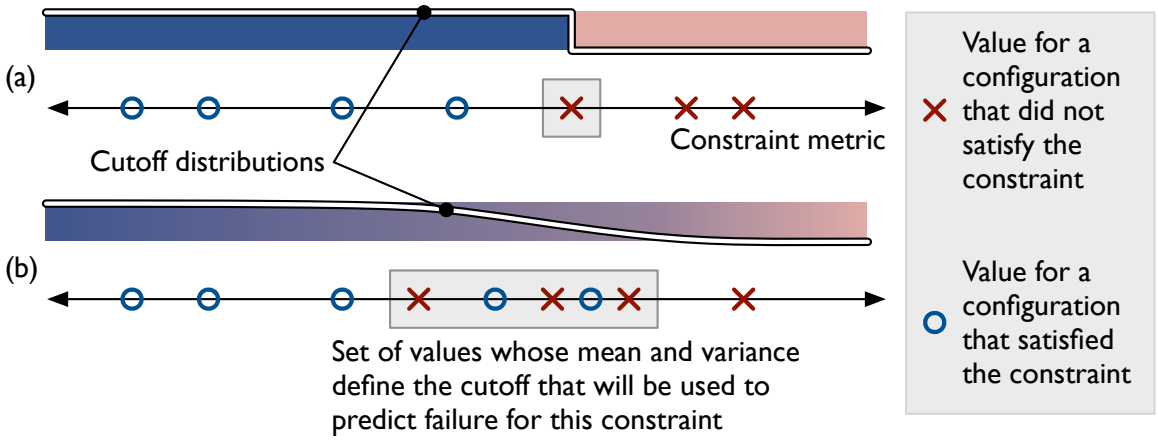


Figure 6.7: Two examples of setting the cutoff value for some constraint. The portion of a candidate's predicted value that is less than the cutoff determines what its predicted probability of passing this constraint will be. Values of this metric for tested configurations are represented as red x's and blue o's. Tested configurations that cannot be said to have passed or failed this constraint (because they failed for some other reason) are not represented here at all. Note that if there is some overlap in the constraint metric between cases that passed and cases that failed, the cutoff will be a distribution, not a scalar value; this is fine: the result of comparing two normal distributions with a greater-than or less-than operator can still be interpreted as a simple probability.

To set the limit for constraint f with proxy metric p, we examine all tested points. If a configuration failed constraint f, its value for metric p is recorded as a failed value. If a configuration succeeded, or even got far enough to prove that it will not fail constraint f, its value for metric p is recorded as a successful value. Configurations that failed in some other way that does not determine whether they would have passed f or not are not recorded.

The failed and successful values are sorted, as indicated in Figure 6.7. The cutoff region is considered to be everything from the lowest failed value up to the lowest failed value that is higher than the highest successful value. In the special case of no overlap between successful and failed values, the cutoff region is a single value. The cutoff is then computed as the mean and variance of the values in the cutoff region. Since the system is already using

normal distributions to model the predictions for all real values, it is completely natural to compare this distribution with the IR size prediction distribution to compute the probability of hitting the compiler time limit.

There are many other strategies that could be used to predict the probability of a candidate satisfying all constraints. For example, classification methods like support vector machines (SVMs[SS01]) or neural networks could prove effective. The classical versions of these methods produce absolute predictions instead of probabilistic predictions, but they have been extended to produce probabilistic predictions in a variety of ways. Also, the intermingling of successful and failing configurations (as opposed to a clean separation between the classes) is a challenge for some conventional classification methods.

6.6 Probabilistic regression analysis

At the heart of the tuning knob search method is probabilistic regression analysis. Regression analysis is the process of predicting values of a real-valued function (usually of a multi-dimensional real space) given a potentially sparse set of training data. Probabilistic regression analysis produces a predicted distribution, as opposed to a single particular value. Classical regression analysis is a more heavily studied problem than the probabilistic variant. However, there are a number of existing methods in the applied statistics and machine learning literature, perhaps the hottest of which in recent years are methods based on Gaussian processes (GPs[RW06]).

While probabilistic regression analysis is a well-studied problem in a variety of contexts, we are not aware of any auto-tuning algorithms that use it. The regression analysis needed for auto-tuning is somewhat different from the conventional applications in predicting natural processes. Most existing approaches to probabilistic regression require a *prior distribution*, which is an assumption about the shape of the function before any training points have been observed. These assumptions are usually based on some formal or informal model of the system being measured. However, auto-tuners generally have no way to know the shape of the function for some program feature.

We must, however, make some kind of assumptions about the underlying functions. Without any assumptions, it is impossible to make predictions; any value would be equally

likely. We designed our own relatively simple probabilistic regression method based on the assumption that local linear averaging and derivative projection are good guides for predicting values of untested configurations.

Several of the subcomponents of the regression analysis developed here, such as distance-weighted averaging in multidimensional space given a sparse set of data points, are themselves well studied problems with a diverse literature of good solutions. I have used existing approaches to these subcomponents to make a regression analysis method that I believe matches the needs of tuning knob searching well.

To keep it as clear as possible, the initial description of the complete tuning knob search method uses simplistic implementations for some subcomponents. More sophisticated alternatives are described in Section 6.9.

Throughout this section we use one-dimensional visualizations to illustrate the mathematical concepts. The math itself generalizes to an arbitrary number of dimensions.

6.6.1 Averaging tested values

The first step in calculating a candidate's distribution is a local linear averaging. For some candidate point $\vec{p}$, $\text{interp}_{\vec{p}}$ is the weighted average[7] of the values of the points that are *neighbors* of $\vec{p}$, where the weight for neighbor $\vec{n}$ is the inverse of the *distance* between $\vec{p}$ and $\vec{n}$. This model has two components that require definition: distance and neighbors.

Distance

The distance between two points is composed of individual distances between two settings on each knob, and a method for combining those individual distances. For continuous and discrete range knobs, we initially assume that the distance between two settings is just the absolute difference between their numerical values. We will discuss scaling these numbers in Section 6.9.5.

We combine the individual distances by summing them (*i.e.*, we use the Manhattan distance). Euclidean distance can be used as well; in our experiments, the impact of the

[7] Any weighted averaging method works (arithmetic, geometric, etc.); we used the arithmetic mean.

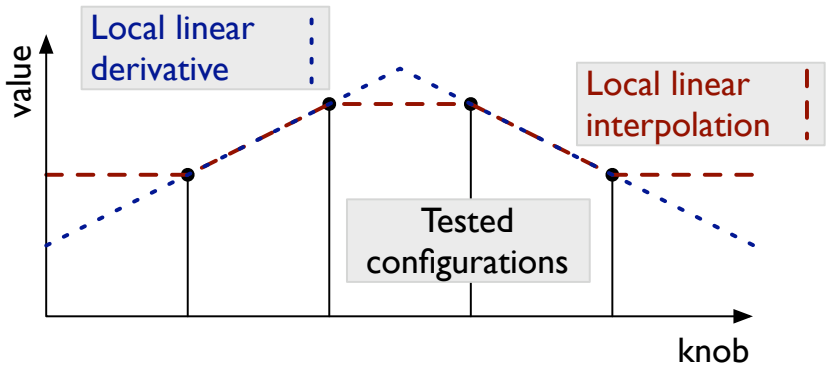


Figure 6.8: A comparison of local linear averaging and derivative projection. Averaging is "safe" in the sense that it never produces predictions outside the range of observed values. However, plain averaging does not follow the trends in the data, and so produces predictions that do not seem intuitively right when the tested values seem to be "pointing" in some direction.

difference between Manhattan and Euclidean distances on final search effectiveness was small.

Neighbors

There are many reasonable methods for deciding which points should be considered neighbors of a given point. For the initial description, we will assume that a point's neighbors in some set are the k nearest points in that set, where we choose k to be 2 times the number of tuning knobs (*i.e.* dimensions) in the application. The intuition for this k value is that if the tested configurations are evenly distributed, most points will have one neighbor in each direction. This definition of neighbors performed reasonably well in preliminary experiments, but has some weaknesses. In Section 6.9.2 we give a more sophisticated alternative.

6.6.2 Derivative-based projection

Averaging is important for predicting the value of a function, but it does not take trends in the training data into account at all, as illustrated in Figure 6.8. In order to take trends

in the data into account, we add a derivative-based projection component to the regression analysis. In a sense, projection is actually serving two roles: (1) it helps make the predictions follow our assumption that functions are piecewise linear; (2) it helps identify the regions where there is a lot of uncertainty about the value of the function.

For each candidate point $\vec{c}$ we produce a separate projection from each of the neighbors of $\vec{c}$. We do this by estimating the derivative in the direction of $\vec{c}$ at each neighbor $\vec{n}$. The derivative estimate is made by using the averaging model to estimate the value of the point ϵ distance from $\vec{n}$ in the opposite direction from $\vec{c}$.

$$\vec{d} = \vec{n} + \frac{\epsilon}{dist(\vec{c}, \vec{n})}(\vec{n} - \vec{c})$$

We use the averaging model to calculate a value for $\vec{d}$, which gives us a predicted derivative at $\vec{n}$.

$$\text{derivative at } \vec{n} \text{ towards } \vec{c} = \frac{\text{value}(\vec{n}) - \text{interp}(\vec{d})}{\epsilon}$$

Finally to get the value for $\vec{c}$ predicted from $\vec{n}$ we project the derivative back at $\vec{c}$.

$$\text{extrap}(\vec{c}, \vec{n}) = \text{value}(\vec{n}) + dist(\vec{c}, \vec{n}) \times \text{derivative}(\vec{n}, \vec{c})$$

A useful property of this projection method is that it takes into account the values of points farther from $\vec{c}$ than its immediate neighborhood; in a sense expanding the set of local points that influence the prediction.

Figure 6.9 illustrates averaging between two tested points and derivative projection from the same points. Three different values are generated for the candidate configuration (the dotted vertical line); the mean and variance of these values become the predicted distribution for this candidate for whatever feature we are currently working with.

6.6.3 Predicted distribution

The final predicted distribution for each candidate point $\vec{c}$ is the weighted mean and variance of its distance-weighted average, and projected values from all its neighbors to compute its predicted value distribution. The selection of the weights is important, and we use a "gravitational" model, where the weight on each projected value is the square of the inverse

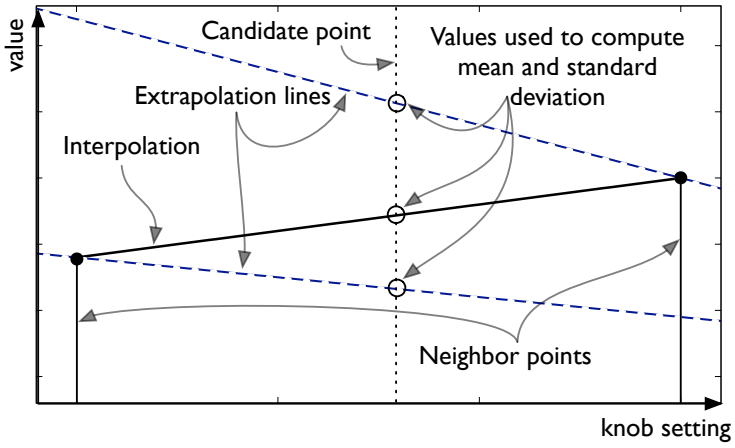


Figure 6.9: The basic ingredients that go into the probabilistic regression analysis used in the tuning knob search. The distribution for a given candidate configuration is the weighted mean and standard deviation of the averaged value between neighboring tested points (black line) and projected values from the slope at the neighbors (dashed blue lines).

distance from the neighbor that we used to calculate that projection ($w_{\vec{n}} = \frac{1}{dist(\vec{p},\vec{n})^2}$). The weight on the averaged value is equal to the sum of the weights on all the projected values. In other words, the averaging is as important as all of the projections combined. So the complete set of weights and values used to compute the predicted distribution is:

$$\left\{\left((\sum_{n\in\text{neighbors}} w_n), \text{interp}(\vec{c})\right)\right\} \cup \left\{\left(w_n, \text{extrap}(\vec{c},\vec{n})\right) \middle| n \in \text{neighbors}\right\}$$

Observe that the variance of this set will be large when the values projected from each of the neighbors are different from each other and/or the averaged value.

Figure 6.10 shows what the predicted distributions would look like for the whole range of candidates between two tested points.

6.6.4 Target quality

Initially we assume that the target is simply the quality of the best configuration found so far; the high-water mark. Figure 6.11 illustrates how candidates' quality predictions are compared with the target. In this example, the highest quality tested point is not shown.

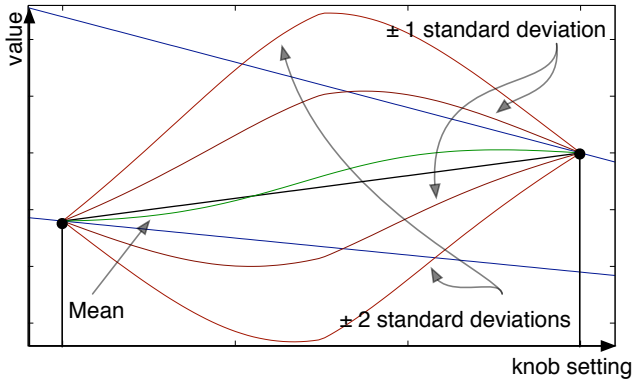


Figure 6.10: An illustration of the distributions that produced by the regression method presented here. Notice that the variance is much higher farther away from tested points. Also the slopes at the neighbors (blue lines) "pull" the mean line up or down. Finally, the "best" configuration to test next depends on the relative importance given to high mean quality versus large variance.

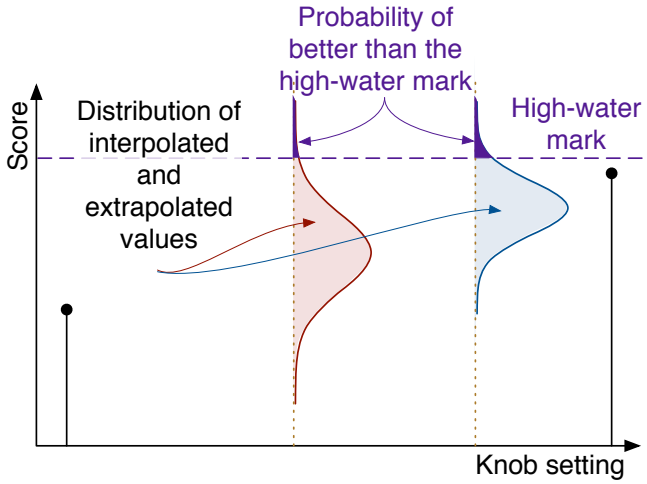


Figure 6.11: An illustration of randomly selected candidate configurations (dotted vertical lines), quality predictions for those configurations (normal distributions), the current target (dashed line), and the probability of a candidate being better than the target (dark portions of the distributions).

6.7 Derived features

An important part of the tuning knob search method is that predictions for sensors (features for which raw data is collected during configuration testing) can be combined in a variety of ways to make derived feature predictions. A very simple example of a derived feature is the product of program run time and energy consumption. It is possible to compute a run time-energy product value for each test point, and then run the regression analysis directly on those values. However, by making predictions independently on the more primitive values we can sometimes make significantly more accurate predictions.

A slightly more complex example of how derived features can be useful is an application with two sequenced loops nested within an outer loop. We can measure the run time of the whole loop nest as a single sensor, or we can measure the run time of the inner loops separately and combine them into a derived feature for the run time of the whole loop nest. If the adjustment of the tuning knobs in this application trade off run time of the two inner loops in some non-trivial way, it is more likely that we will get good predictions for the individual inner loops. The individual effects of the knobs on the inner loops are conflated together in the run time of the whole loop nest, which makes prediction harder.

An example of a derived feature that is used in our experiments is the proxy metric for compiler time limit violations. The proxy metric combines a number of measures of intermediate representation size and complexity; each measure is predicted in isolation, then the predictions are combined using derived features.

Each mathematical operator that we would like to use to build derived features needs to be defined for basic values (usually simple) and distributions of values. The simplest operators are addition and subtraction. The sum of two normal distributions (for example the predicted run times for the two inner loops in our example above) is a new normal distribution whose mean is the sum of the means of the input distributions and whose standard deviation is the sum of the input standard deviations. This definition assumes that the input distributions are independent, which is a simplifying assumption we make for all derived features.

Multiplication and division are also supported operators for derived features. Unfortu-

nately, multiplying and dividing normal distributions does not result in distributions that are precisely normal. However, as long as the input distributions are relatively far from zero, the output distribution can be closely approximated with a normal distribution. We use such an approximation, and it is currently left to the user (either the programmer or the compiler writer) to decide when this might be a problem.

There are also comparison operators, like less-than and greater-than, that take normal distributions as inputs and produce a single probability as an output. The probability of a number drawn at random from distribution X being smaller than a number drawn at random from distribution Y is (as usual, assuming independence):

$$\begin{aligned} P(X < Y) &= P(X - Y < 0) \\ &= P(Z < 0), \text{ where } \mu_Z = \mu_X - \mu_Y \text{ and } \sigma_Z = \sigma_X + \sigma_Y \\ &= P(Z' < \mu_Y - \mu_X), \text{ where } \mu_{Z'} = 0 \text{ and } \sigma_{Z'} = \sigma_Z \\ &= P(Z'' < \frac{\mu_Y - \mu_X}{\sigma_X + \sigma_Y}), \text{ where } Z'' \text{ is the standard normal distribution} \end{aligned}$$

The probability of a number drawn at random from the standard normal distribution being below some given constant can be computed using standard methods.

Probabilities can be combined with Boolean operator features (AND, OR, ...). For example, the probability of the conjunction of two independent events is the product of the probabilities of the individual events.

Finally, the tuning knob system has "aggregator" functions, like minimum and maximum, that are computed over all the configurations that have been tested so far. Aggregator functions are essential for computing features like the quality high-water mark.

6.8 Complete basic tuning knob algorithm

The complete basic tuning knob algorithm is shown in Figure 6.12.

6.9 Enhancements

As described so far, the search method worked reasonably well in our preliminary experiments, but we found a number of ways to improve its overall performance or its robustness.

```
1  Basic tuning knob search
2      Initialize T, the set of results, to empty
3      Select a configuration at random, test it, add results to T
4      while (termination criteria not met)
5          Compute quality target value
6          Do pre-computation for regression analysis (e.g. build neighborhood)
7          Repeat N times (100 in our experiments)
8              Select an untested configuration c at random
9              Perform regression analysis at point c for all sensors
10             Evaluate derived features at point c
11             pSuccess ← 1
12             ∀f ∈ failure modes
13                 pSuccess ← min(pSuccess, P(feature linked to f))
14             score for c = P(quality(c) > target) × pSuccess
15         Test candidate with highest score, add results to T
```

Figure 6.12: The complete basic tuning knob search algorithm.

The main quantitative result in the evaluation section will show the large difference between our search method with all the refinements versus using the approach that treats all failing configurations as having a very low quality. Compared to the large difference between sophisticated and simplistic failure handling, the refinements in the following sections have a relatively small impact. They are described here for completeness; quantitative evaluation of their individual impacts is left to future work.

6.9.1 Target quality

Given a predicted quality distribution for each candidate in a set, it is not immediately obvious which is the best to test next. This is true even if we ignore the issue of failures entirely. Some distributions have a higher mean and smaller variance, whereas some have a

larger variance and lower mean. This is a question of how much "risk" the search algorithm should take, and there is no simple best answer. The strategy we use is to compute the probability that each candidate's quality is greater than some target; in other words, the likelihood that a value selected at random from a candidate's quality distribution is higher than the target.

The simplest method for choosing the target that worked well in our experiments is using the maximum quality over all the successful configurations that have been tested so far. There is no reason that the target has to be exactly this high-water mark, though, and adjusting the target is a very effective way of controlling how evenly distributed the set of tested points is. The evenness of the distribution of tested points is an interesting metric because the ideal distribution of tested points is neither perfectly evenly distributed nor too narrowly focused. Distributions that are too even waste many searches in regions of the configuration space that are unlikely to contain good points. Distributions that are too uneven run a high risk of missing good points by completely ignoring entire regions of the space.

To keep the set of tested points somewhat evenly distributed the target is adjusted up (higher than the high-water mark) when the distribution is getting too even and adjusted down when the distribution is getting too uneven. Higher targets tend to favor candidates that have larger variance, which are usually points that are farther from any tested point, and testing points in "empty space" tends to even out the distribution.

There are many ways to measure the evenness of the distribution of a set of points. We are currently using the *coefficient of variation* of the distances between all neighbor pairs. The coefficient of variation of a set of numbers is the standard deviation of the set divided by the mean. If the tested points are very evenly distributed, the coefficient will be close to zero. If the points are very irregularly dispersed, the coefficient of variation will be close to 1. To take edge effects into consideration, the extreme corners of the space are included as pseudo-points for the coefficient of variation calculation. Other ideas about how to compute a coefficient of "uniformity" or "empty space" include [FLC03] and [JXHX02].

Two more concepts affect target quality adjustment. The first is the percentage of the tested points that succeeded. The motivation for including this is that if too many tested

points are failing, one of the possible reasons is that the search is choosing candidates with variances that are too high.

The last factor that goes into my target adjustment is the average across all the candidate points of the standard deviation of their quality prediction. This may seems unnecessarily byzantine; the point is just to get the target correction into the right units and the right sort of scale to have an influence on which candidate is selected.

With this approach to adjusting the quality target the complete probability formula becomes:

$$P(\text{quality}(c) > q_t) \times \min_{N \in \text{constraints}} \big(P(c \text{ satisfies constraint}_N)\big)$$

$$\begin{aligned} q_t \;=\; & \max_{t \in \text{tested}} \big(\text{quality}(t)\big) + \\ & \big(\text{coefficient of variation}(\text{Distances between neighbor pairs}) \times \\ & \frac{\#\text{ of successful tests}}{\#\text{ of tests}} \times \\ & \operatorname*{Avg}_{c \in \text{candidates}} (\text{Standard deviation of quality prediction of } c)\big) \end{aligned}$$

The motivation for exploring target adjustment was that in some of our early experiments with the simple high-water mark, the distribution of tested configurations seemed far too tightly clustered.

6.9.2 Neighborhoods

When the distribution of a set of points is fairly uneven, the simple k-nearest and radius δ hypersphere definitions of neighbor pairs (illustrated in Figure 6.13) do not capture some important connections. In particular, points that are relatively far apart but do not have any points in between them might not be considered neighbors, because there are enough close points in other directions.

To get a better neighbor connection graph, we use a method similar to the elliptical Gabriel graph described in [PSC06]. Two points are considered neighbors as long as there does not exist a third point in "region of influence" between them. There are many reasonable options described in the cited paper for defining the shape of the region of influence. Our implementation, for which there is pseudocode directly below, defines the region of

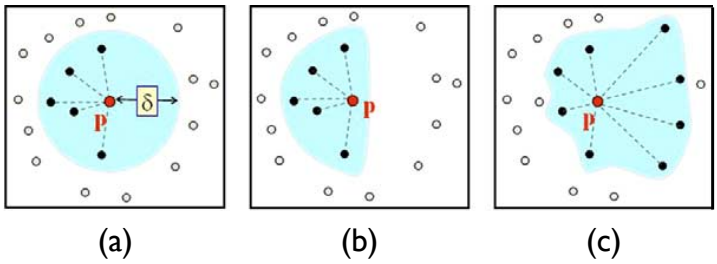


Figure 6.13: The mathematically simple methods for defining neighbor graphs can produce unsatisfactory results. (a) illustrates the radius δ hypersphere method; (b) illustrates the k-nearest method (with k=5). In both cases the set of neighbors is highly skewed. (c) shows a more desirable neighbor graph. Graphic borrowed from [PSC06].

influence between two points as the intersection of two spheres around the two points and an ellipsoid with the two points at the foci. The aspect ratio of the ellipse is defined by a parameter ("ellipseConst") that is set to 1.1 for all experiments. The distance constraints are illustrated in Figure 6.14.

areNeighbors(x_1,x_2,points)
    $d_{1,2}$ = distance(x_1, x_2)
    $\forall x_3 \in$ (points $\setminus \{x_1, x_2\}$)
        $d_{1,3}$ = distance(x_1, x_3)
        $d_{2,3}$ = distance(x_2, x_3)
        if $\big((d_{1,3} < d_{1,2}) \wedge (d_{2,3} < d_{1,2}) \wedge (d_{1,3} + d_{2,3} <$ ellipseConst $\times\, d_{1,2})\big)$
            return false
    return true

The naïve approach to building the neighborhood graph with this method scales badly with the number of tested configurations. For every pair of configurations, every other configuration must be checked to see if it is between, which makes the total running time $O(N^3)$. We have run searches up to about 100 tests, and not found the neighborhood graph building run time to be a problem. For scaling the tuning knob search up to larger numbers of tested configurations, there are more efficient methods for building the graph given in

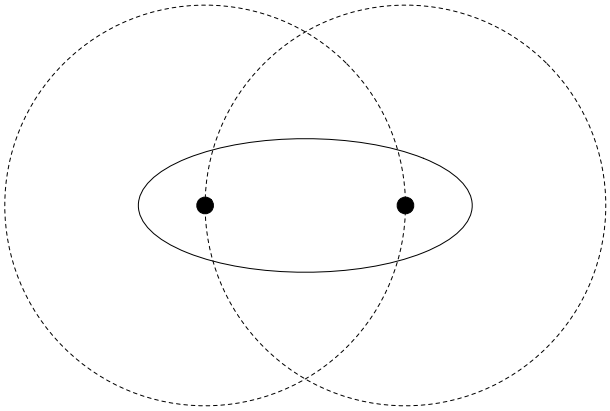


Figure 6.14: An illustration of how we compute whether some third point is between two other points. A third point must lie in the intersection of the two circles and the ellipse to be considered between the first two (hyperspheres and hyperellipse in N-dimensional space).

[LKSK08].

6.9.3 Boundary conditions

In early experimentation, the sensor predictions for configurations outside the convex hull of tested points tended to have unreasonably high variance. To counteract this problem, we add pseudoconfigurations outside the space of legal values by mirroring the actual tested points across the boundary of the tuning knob space (not the convex hull of tested points). For example, if there are two knobs with ranges [1,10] and [2,15] a real point at (7,11) will have four mirrored points: (1-(7-1),11), (10+(10-7),11), (7,2-(11-2)), (7,15+(15-11)), which equal (-5,11), (13,11), (7,-7), (7,19). These points are treated as regular training points by the regression analysis, and they reduce the predicted variance of the predictions on the periphery of the space.

6.9.4 Combining constraints, part II

In experimenting with this search system we found that there were situations where the algorithm would test "too many" points that ended up failing for a particular reason, and

other situations where the algorithm would stay "too far away" from a region that had some failures. To even out these imbalances we added a negative feedback loop mechanism to the constraint probability calculations. The system keeps track of what percentage of all tested points have failed because of violating each constraint. If this percentage is low, the probability of failing for that reason is scaled down; if the percentage is high, the probability is scaled up.

This mechanism allows the constraint prediction formulas to be only "relatively" accurate rather than "absolutely" accurate. As long as the formula correlates reasonably well with whether a point will actually violate a constraint or not, this feedback mechanism will use the actual search data to scale up or down the actual probabilities to achieve a "good" balance between testing successful and failing configurations. This makes the overall probability that a configuration is the best to test next:

$$P(\text{quality}(c) > q_t) \times \min_{N \in \text{constraints}} \Big(S_N\big(P(c \text{ satisfies constraint}_N)\big) \Big)$$

$$\begin{aligned} S_N(p) &= p^{\left(1 - \text{Min}\left(\beta, \frac{\#\text{ points that violate constraint } N}{\#\text{ points tested}}\right)\right)} \\ q_t &= \max_{t \in \text{tested}} \big(\text{quality}(t)\big) + \\ &\quad \big(\text{coefficient of variation}(\text{Distances between neighbor pairs}) \times \\ &\quad \frac{\#\text{ of successful tests}}{\#\text{ of tests}} \times \\ &\quad \operatorname*{Avg}_{c \in \text{candidates}} (\text{Standard deviation of quality prediction of } c)\big) \end{aligned}$$

The β cap helps the search perform reasonably in pathological situations, like early in a search if all of the tested configurations failed for the same reason. In our experiments, β is 0.9.

6.9.5 Distance scaling

Intuitively, not all tuning knobs are equally "important". What this means for our search process is that we want to be more "careful" to ensure that we get good distribution of points in some dimensions than others. We could accomplish this goal by scaling up the raw distances in the dimensions that we estimate are more important.

The way scaling works is that for each tested point, we want to get an estimate of the derivative of the score function in each dimension. We do this by using distance-weighted averaging to get an estimated value at points $\pm\epsilon$ away from each tested point in that dimension. The absolute difference between the predicted value ($\pm\epsilon$ away) and the actual value of the tested point, divided by ϵ, gives a slope estimate for that dimension.

For each dimension (knob) we then average the slope across all the tested points to get an average importance factor for that knob. From that time on, the distance in each dimension gets scaled up by the ratio of that dimension's importance factor to the least important dimension's factor. Note that when using a neighborhood-based averaging method, a point's neighbors can change when the dimension scaling factors change. One could imagine iterating the distance scaling process and the neighborhood determination process until it reaches a fixed point, and no more neighborhood changes happen. We have found one iteration to be sufficient.

6.9.6 Running multiple tests simultaneously

So far we have assumed that only one test runs at a time, and when it completes, a new candidate point is selected. However, given the high computational demands of compilation for some accelerators, some users will want to use a cluster of development machines to test multiple points simultaneously.

In a production setting, adapting the search process for parallel tests is relatively easy. Whenever a machine is free to test a new point, there will be a number of configurations that have been selected but not completed yet. A reasonable approach to handling inflight configurations is to assume that for every program feature, the point's value is whatever the mean of its prediction was when it started testing. Because the "ersatz score" of the inflight points matched the model prediction perfectly, the variance for sensor predictions near such points will be low, which discourages searching close to inflight points—a good thing.

6.9.7 Adding randomness to candidate selection

The basic method of selecting the candidate configuration with the highest heuristic score works reasonably well. However, no predictive model can be perfect, and all behave poorly on some training data. As a defense against getting trapped in pathological situations, we have experimented with adding an additional layer of randomness at the end of the candidate selection process. After each candidate has been assigned a score, we could select a candidate randomly, with higher scoring candidates given a higher probability of being selected.

6.9.8 Termination criteria

The search process can terminate at any time (preferably after at least one successful configuration has been found!). Reasonable termination conditions include testing a fixed number of points, testing for a fixed amount of wall clock time, testing until a certain number of points have been tried without finding any improvement. All the experiments reported in this chapter performed 50 tests per search.

6.10 Evaluation

Comparing the tuning knob search against existing auto-tuning approaches is problematic because the main motivation for developing a new search method was handling hard-to-predict failures, and we are not aware of any other auto-tuning methods that address this issue. As evidence that our failure handling is effective, we compare the complete tuning knob algorithm against the same algorithm, but with the constraint/failure prediction mechanisms turned off. Points that fail are simply given the quality value of the lowest quality configuration found so far.[8]

As a basic verification that the tuning knob search algorithm selects a good sequence of configurations to test, we also compare against pseudorandom configuration selection. As mentioned in the previous chapter, pseudorandom searching is a common baseline in the

[8]We also tried making the score for failing points lower than the lowest found so far. The results were not significantly different.

auto-tuning literature. Other baselines that are used in some published results include hill-climbing style searches and simulated annealing-style searches. These kinds of algorithms could be combined with trivial failure handling, but we chose not to perform these comparisons because they would not shed light on the central issue of the importance of handling failures intelligently.

Within the structure of the tuning knob search algorithm we can imagine investigating the effectiveness of the core probabilistic regression analysis for sensors, and comparing it with other regression methods. While this is an interesting direction for improvement of the overall search performance, it is again secondary to the main issue of failure handling. We leave such investigations of the regression analysis itself to future work.

We tested four applications written in Macah and targeted coarse-grained reconfigurable arrays in the Mosaic infrastructure. The applications were chosen for their interesting use of tuning knobs. To show that the tuning algorithm adapts to different architectures we used four different specific architectures in the same family by varying two parameters: the number of clusters (small or large) and the number of memories per cluster (few or many).

6.10.1 The applications

To give a sense for the shapes of the tuning spaces, below are plots of all the performance and failure data gathered for each application. These plots include data from experiments with many different search methods, so the distribution of tested configurations is not meaningful. Each application has one plot for each architecture. The meaning of the symbols in the plots are given in the following table. See Section 3.5 for a definition of initiation interval.

Symbol	Meaning
Black dot	Not tested
Green star	Max initiation interval failure
Empty blue square	Data memory or I/O failure
Red X	Compiler timeout failure
Filled square	Color indicates normalized quality (1 is best)

The finite impulse response (FIR) filter application, which was discussed in detail earlier

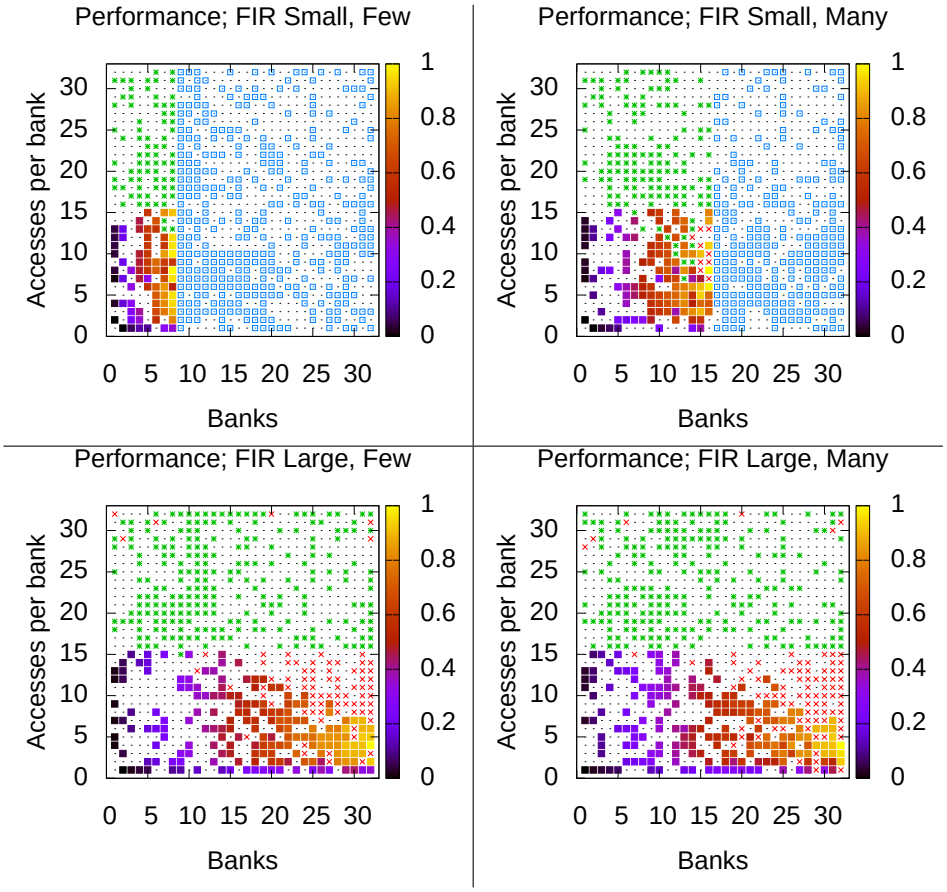


Figure 6.15: Normalized performance and failure modes for the FIR filter. Orange/1 is the highest performance setting; Black/0 is the lowest performance setting. Small/Large and Few/Many refer to the architecture variants that we experimented with.

in this chapter, has a knob that controls the number of banks into which the coefficient and input buffer arrays are broken. The more banks, the more distributed memories that can be used in parallel. The second knob controls the number of accesses to each bank per iteration. Performance should increase with increasing values of both knobs, because more parallel arithmetic operations are available.

As you can see in Figure 6.15, both data memory and initiation interval failures are common; more so in the smaller architectures. The number of memories clearly limits the banks knob, which results in the large regions of failures on the right side of the plots for

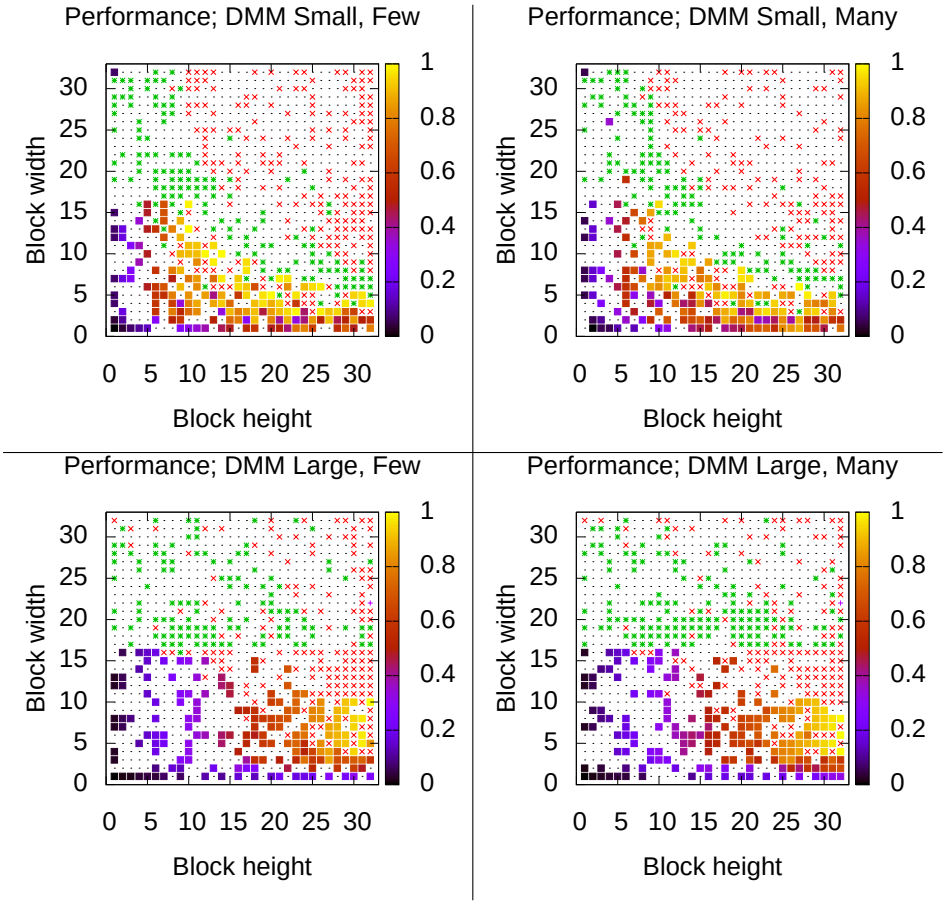


Figure 6.16: Normalized performance and failure modes for dense matrix multiplication. Orange/1 is the highest performance setting; Black/0 is the lowest performance setting. Small/Large and Few/Many refer to the architecture variants that we experimented with.

the smaller architectures. The number of parallel accesses to an array is limited by the maximum initiation interval of the architecture, which creates the large regions of failure toward the top of the plots. The larger architectures show an interesting effect, when both knobs are turned up relatively high the compiler begins to hit the time limit because the application is just too large to compile. This creates the jagged diagonal line of failures.

The dense matrix multiplication implementation has one level of blocking in both dimensions in the SUMMA style[vW97]. SUMMA-style matrix multiplication involves reading stripes of the input matrices in order to compute the results for small rectangular blocks of

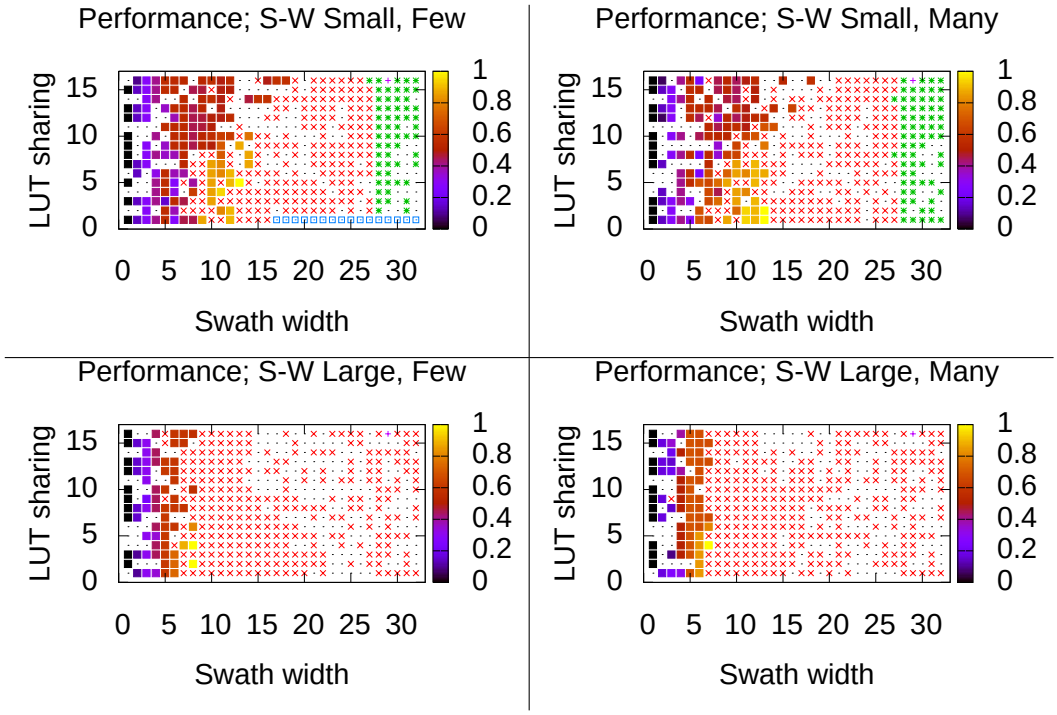


Figure 6.17: Normalized performance and failure modes for Smith-Waterman. Orange/1 is the highest performance setting; Black/0 is the lowest performance setting. Small/Large and Few/Many refer to the architecture variants that we experimented with.

the output matrix, one at a time. The tuning knobs control the size of the blocking in each dimension.

Like FIR, the highest performance configurations are towards the top right of the plots, but failures are a major factor. Notice that failing configurations are intermixed with successful configurations, which would make it very hard to make a formal model to predict precisely which configurations will fail.

The Smith-Waterman (S-W) implementation follows the common strategy of parallelizing a vertical swath of some width. One of the tuning knobs controls the width of the swath and the other controls the number of individual columns that share a single array for the table lookup that's used to compare individual letters.

The S-W results show an interesting and counterintuitive effect: the range of tuning knob

settings that work is actually smaller on the larger architecture than it is on the smaller architecture. The reason is that something about the S-W application is more challenging for the backend of the compiler and the larger architecture puts more stress on placement and routing, causing compilation to timeout in more cases.

The 2D convolution implementation has one knob that controls the number of pixels it attempts to compute in parallel, and another that controls the width of the row buffer (assuming that all of the rows do not fit in memory simultaneously). Of the applications and configurations that we have tested, the convolution setup had one of the most challenging shapes.

For all the applications, the highest quality configurations are directly adjacent to, and sometimes surrounded by, configurations that fail. This supports the assertion that in order to have any hope of finding the highest quality configurations a tuning method for accelerators needs an approach to failure handling that is at least somewhat sophisticated. For example, a search that had a strong preference for testing points far away from failures would clearly not perform particularly well.

6.10.2 Results

The experimental validation of our tuning strategy involved performing tuning searches for each of the application/architecture combinations with a particular candidate selection method. The three main methods compared were the full tuning knob search, a purely random search and the tuning knob search with the trivial approach to failures. In all cases the termination criterion was 50 tested configurations. All the search methods have some form of random behavior, so we ran each experiment with 11 different initial seeds; the reported results are averages and ranges across all initial seeds.

Figure 6.19 shows the summary of the search performance results across all applications and architectures. To produce this visualization, the data for each application/architecture combination are first normalized to the highest quality achieved for that combination across all experiments. For each individual search we keep a running maximum, which represents the best configuration found so far by that particular search. Finally we take the average

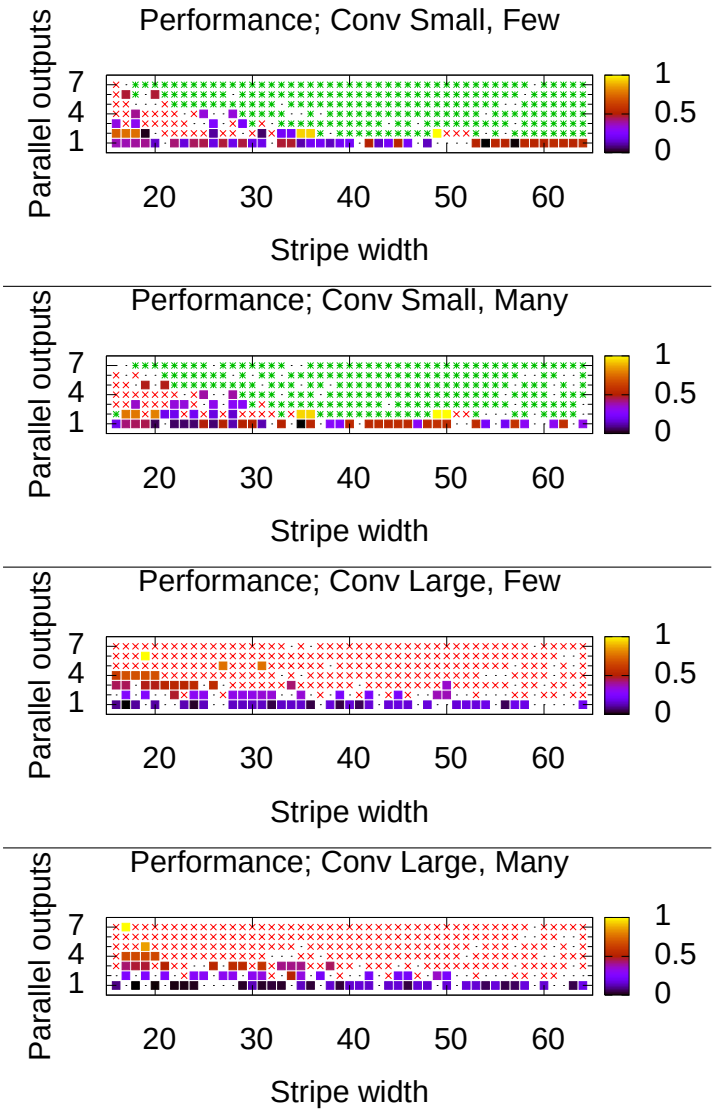


Figure 6.18: Normalized performance and failure modes for 2D convolution. Orange/1 is the highest performance setting; Black/0 is the lowest performance setting. Small/Large and Few/Many refer to the architecture variants that we experimented with.

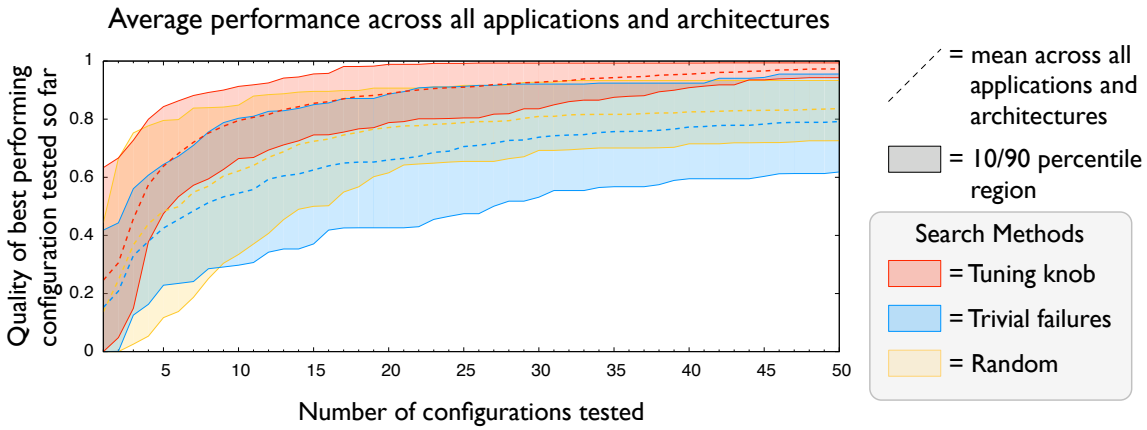


Figure 6.19: Given a particular number of tests, the complete tuning knob strategy clearly finds higher quality configurations on average than either the pseudorandom method or the method without failure prediction.

and 10th/90th percentile range across all application/architecture/initial seed combinations.

The headline result is that the tuning knob search method is significantly better than the other two methods. For almost the entire range of tests the 10th percentile quality for the tuning knob search is higher than the mean for either of the other two methods.

Interestingly, it seems that the purely random search does better than the tuning knob search with the trivial approach to failures. Our intuition for this result is that the tuning knob search that assigns a constant low quality to all failing points does not "understand" the underlying cause for the failures and chooses to test too many points that end up failing. To reemphasize, without failures in the picture at all, some completely different search strategy might be better than the tuning knob search. However, it is very interesting that for these applications that have a large fraction of failing configurations there is a very large gap between a reasonably good search method that makes smart predictions about failure probability and one that uses essentially the same predictions, but treats failures trivially.

Figure 6.20 shows the same search results summarized in a different way. This figure has the axes swapped compared to Figure 6.19, which shows the number of tests required

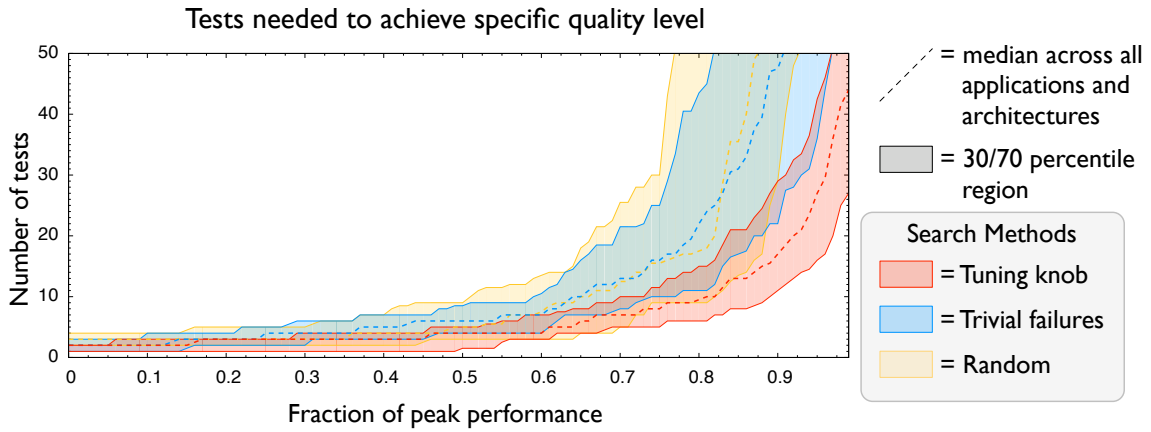


Figure 6.20: Number of tests required to achieve a specific fraction of peak performance, averaged across all application/architecture combinations.

to achieve a specific fraction of peak performance. To produce this plot, for each individual search (application/architecture/initial seed) we calculated how many tested configurations it took to achieve a certain quality level. We then aggregated across all the applications, architectures and initial seeds for each search strategy and computed the 30th percentile, median and 70th percentile[9] at each quality level. The essential difference between these two plots is which dimension the averaging is done in.

The important result that this plot shows clearly is that for the interesting quality range, from about 50% to 90% of peak, the random and trivial failure strategies take at least twice as many tests on average to achieve the same level of quality. After 50 tests, the median search for both of the less good strategies are still reasonably far from the peak, and without running many more tests it is hard to know how many it would take to approach peak performance.

It is interesting to observe that the mean performance after 50 tests for the trivial failure strategy is below the mean performance for the random strategy (Figure 6.19). However, when we look at Figure 6.20, we see the the performance level at which the median number

[9]The reason that this visualization has a 30/70 range and the Performance has 10/90 is that the data has more "spread" in one dimension than the other. Look at Figure 6.19 and imagine a horizontal line at any point, and notice how much wider a slice of the shaded region it is than a vertical slice.

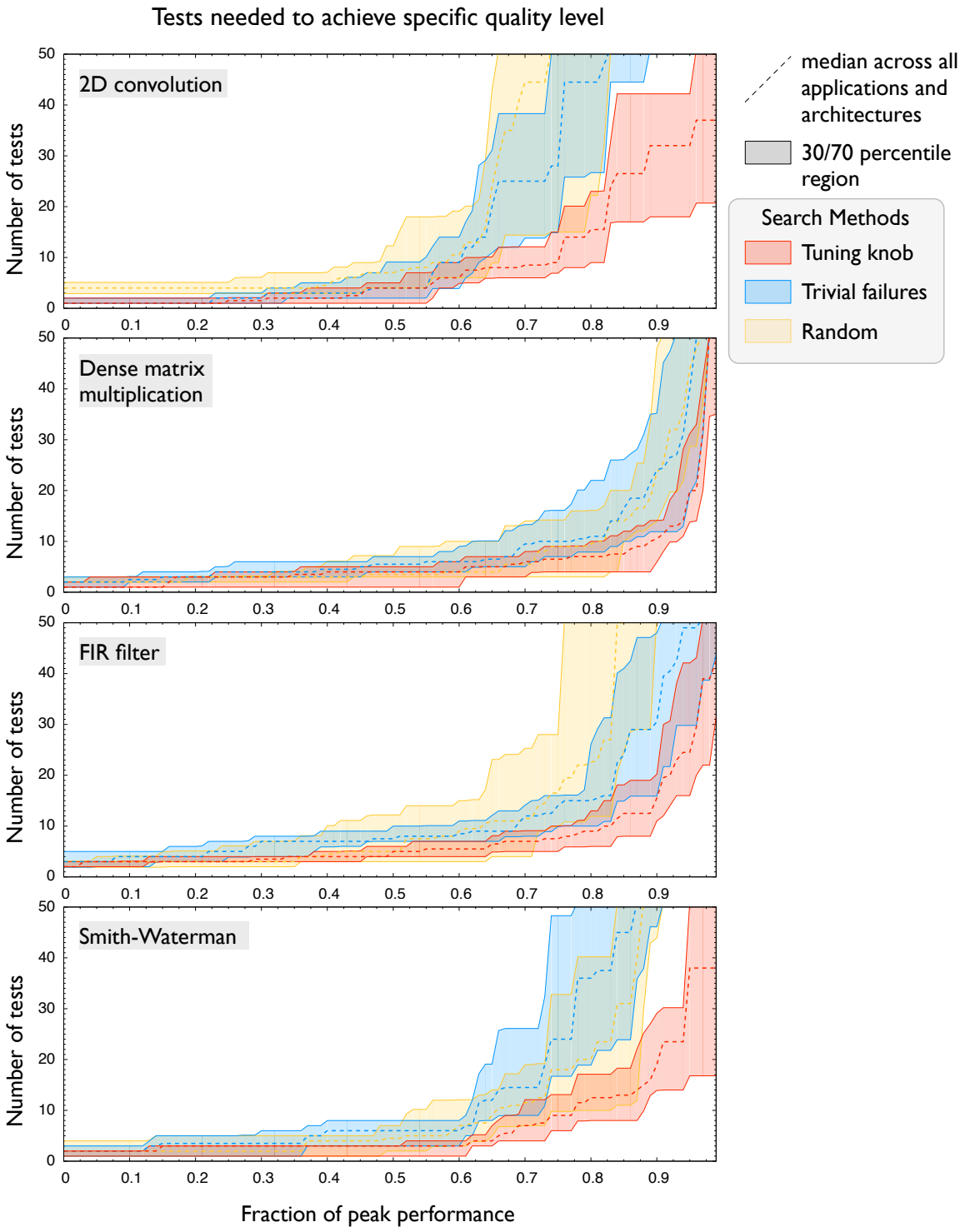


Figure 6.21: Number of tests needed broken up by application.

of tests required is 50 is higher for the trivial failure strategy than the random strategy. The reason for this apparent contradiction is the difference between mean and median. The trivial failure strategy had a relatively small number of searches that ended with a very low quality configuration, which drags down the mean more than it drags down the median.

Figure 6.21 shows the number of tests as a function of quality data broken up by application. We will comment on two interesting features of the data. First, the application for which the tuning knob search has the biggest advantage over both the random search and the trivial failures search is 2D convolution. This application has the most complex boundary between failing and successful configurations. The proxy metrics do a reasonably good job of identifying the boundary, which means that the full tuning knob search spends the majority of its tests in the most interesting region of the space.

Second, the only application for which the trivial failure search does better than the random search is the FIR filter. This application has the simplest regions of failure; for the smaller architectures, the separations are perfectly linear functions of the tuning knobs.

These two cases support the conclusion that the more complex the separation between configurations that satisfy all constraints and those that do not, the more important it is to model the causes of failures explicitly. Conversely, for applications that do not have failures at all, or have failures that can be predicted easily, existing single-factor optimization techniques may be sufficient. More data for individual application/architecture combinations can be found in Appendix A.

As mentioned earlier, we have run some preliminary experiments on the contribution of individual features of the tuning knob search algorithm, like the target quality adjustment and failure probability scaling. Turning these features off has a small negative impact on search quality, but the effect is much smaller than using the trivial approach to failures. We leave a more detailed analysis and tuning of the tuning algorithm to future work.

6.11 Summary

Tuning applications to specific architectures is critical for parallel coprocessor accelerators. Defining the space of possible configurations can be achieved in a number of different ways; adding tuning knobs to a programming language partitions the problem nicely. The pro-

grammer provides the human intelligence about what kind of high level transformations to apply to a program and the compiler does the grunt work of discovering which configurations work best on a particular target architecture.

Tuning for accelerators is harder than tuning for conventional processors because it is much more common for configurations to fail entirely, which makes the quality function only partially defined. It is possible to treat all failing tests as simply having very low quality. However, our experiments with such an approach performed very poorly on average. These results suggest that it is critical for tuning methods for accelerators to make predictions related to both quality and likelihood of failure.

The tuning knob search developed in this chapter uses probabilities and probabilistic distributions to model all its predictions, which makes combining multiple factors relatively simple. The search algorithm also has a couple of negative feedback loops that ensure the distribution of tested points is neither too even nor too uneven, and that not too many points fail for a particular reason.

Chapter 7

RELAXED OPERATIONAL SEMANTICS FOR DYNAMIC STREAMING LANGUAGES

C-like languages for accelerators have a serious language definition problem. The parts of programs that are accelerated (kernels) are generally written as sequential loop nests. The compiler, perhaps with hints from the programmer, uses a number of loop transformations, like tiling and pipelining, to parallelize the kernels. These loop transformations can dramatically reorder the operations in the kernel, relative to the sequential interpretation. The compiler can use easy, conventional analyses to avoid reorderings that would violate dependencies through local variables and control flow. However, non-local dependencies are a bigger challenge.

Non-local dependencies include reads and writes through global pointers, sends to and receives from inter-thread streams, and general system I/O. In this chapter we focus on dependencies through streams because that is a common mechanism for both inter-thread communication and global memory access in C-like languages for accelerators. At the end of the chapter there are a few comments on other kinds of non-local dependencies.

The problem with non-local dependencies is that the language definition either has to guarantee that they will be respected, which forces the compiler to prove that all transformations preserve non-local dependencies, or explicitly relax the non-local behavior of kernels, which makes it significantly harder to reason about the behavior of programs. This dilemma has many similarities with the problem of deciding what the memory model should be for multithreaded shared memory programming languages: sequential consistency is the definition that programmers are comfortable with, but most real implementations have to relax the memory model in some way for performance reasons.

The solution that we propose for C-like languages with streams is analogous to the Java [MPA05] and C++ [BA08] memory models. In those languages, as long as a program

does not have a data race, it will execute as if the underlying machine were sequentially consistent. Programs that do have data races can go wrong in unpredictable ways.[1] In Macah, programs that are well-behaved in ways that are defined below will execute as if all stream operations in all threads execute in program-order.

As a very simple example of the kinds of problems we will address in this chapter, consider the following program. When the stream operations are executed in program-order, it works fine. <? is the receive operator and <! is the send operator; the ‖ notation indicates that the two blocks of code are run as parallel threads.

t_1		t_2
s_1 <! 42	‖	for (i = 0; i < 10; i++) {
for (i = 0; i < 10; i++) {		x_2 <? s_1
x_1 <? s_2		s_2 <! x_2
s_1 <! x_1		}
}		

The program passes the value 42 back and forth between the two threads, and then finishes with 42 in stream s_1's buffer. Now we assume that the code in t_2 is a kernel to which we apply software pipelining. The result of such a transformation would look like:

t_1		t_2
s_1 <! 42	‖	x_2 <? s_1
for (i = 0; i < 10; i++) {		for (i = 0; i < 9; i++) {
x_1 <? s_2		x_3 <? s_1
s_1 <! x_1		s_2 <! x_2
}		x_2 := x_3
		}
		s_2 <! x_2

This transformation is perfectly legal if we consider only the local dependencies in thread t_2. However, the stream operation ordering of t_2 has been altered, which can change how it interacts with other threads. In particular, t_2 now receives twice from s_1 before it sends to s_2. Since t_1 sends only once before its first receive, the program is now guaranteed to deadlock.

---

[1] There are many more details in the Java and C++ memory models; this is a reasonable first-order approximation.

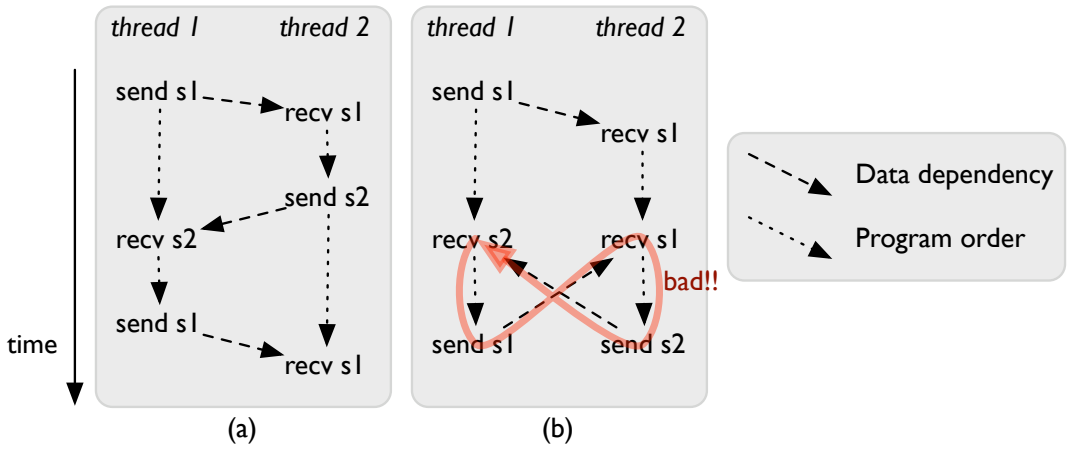


Figure 7.1: Traces of the simple loop in the body of the text. (a) shows a trace of the unoptimized version that works correctly. (b) shows a trace after thread 2 have been pipelined. Notice the causality loop in the dependencies and program order constraints.

The problem in this example is that the first receive in t_1 depends on the first send in t_2, the second send in t_1 is ordered after the first receive, the second receive in t_2 depends on the second send in t_1, and the first send in t_2 is ordered after the second receive. This cycle of dependencies and ordering relationships means no progress can be made, but the compiler does not see that its pipelining optimization created the cycle because it did not analyze the inter-thread dependencies. Figure 7.1 illustrates this problem.

The potential for these kinds of deadlocks do exist in real C-like language implementations for accelerators. The genealogically related languages NAPA C[GS98], Streams-C[GSAK00] and especially Impulse C[PT05] are closely related to Macah. They share several concepts, like streams and pipelined kernels. However, Impulse C simply ignores the interaction between stream sends and receives and loop pipelining. The authors of the Impulse-C book [PT05] observe this fact in Section 4.10 with little further comment.

In order to achieve good performance on accelerators, compilers must be permitted to use transformations like loop pipelining, so we have two options for defining C-level streaming languages:

- A relatively "strong" definition that matches normal intuition about execution order, but forces the compiler to do hard global analyses to ensure that kernel optimizations are safe.
- A relatively "weak" definition that explicitly allows local reordering, but creates possibilities for unexpected results.

For Macah we propose weak semantics that are provably equivalent to the stronger semantics for certain classes of programs.

Rather than trying to specify all legal reordering transformations using an axiomatic style, as is common in the relaxed memory ordering literature[AG95], we follow the style of Boudol and Petri [BP09] and define relaxed operational semantics. These operational semantics can be viewed as an abstract machine that interprets the threads essentially in program order, but uses tricks to make stream operations appear to happen in a different order. We informally argue that the relaxed operational semantics have essentially the same scheduling flexibility that the compiler needs for efficient loop transformation.

7.1 Summary of Results for Non-Language Semanticists

The following sections in this chapter use technical programming language terminology, so in this section we provide a summary of the results in less technical terms. The main contribution of this chapter is a pair of formal definitions for a core Macah-like language; one of the definitions keeps all stream operations strictly in program-order and one has extra language definition machinery to allow reordering of stream operations. The program-order semantics are more intuitive and the relaxed semantics are closer to a realistic implementation.

Though not equivalent in general, we prove that the program-order and relaxed semantics are equivalent for certain classes of Macah programs. This equivalence means that as long as a particular program falls within one of these "well-behaved" classes, programmers are free to reason about it as if all its stream operations happened strictly in program order. This dual-semantics approach is analogous to shared memory multithreaded languages that guarantee that data-race-free programs behave as if the implementation were sequentially

consistent, while explicitly acknowledging that real implementations are not sequentially consistent for all programs.

The kinds of Macah programs for which the program-order and relaxed semantics are equivalent are those that avoid bad cyclic communication patterns through streams. By cyclic communication patterns we mean situations where a thread performs a send on some stream, then later[2] performs a receive that is directly or indirectly dependent on the previous send. We have not yet found a class as general as data-race-free in the shared-memory context, but we do present equivalence proofs for some useful classes of programs. For example, Macah programs with no cyclic communication through streams at all are safe.

The method we use for proving equivalence between different semantics is inspired by [BP09]. The semantics are small-step operational, which means they can be thought of as abstract machines that evaluate programs step-by-step. The proofs show that given an arbitrary sequence of evaluation steps under one semantics for a program that is well-behaved in some defined way, we can find an equivalent sequence of evaluation steps under the other semantics. Sequences of states are considered equivalent if the abstract states that they imply are equivalent. If for all possible sequences of states under each semantics we can find an equivalent sequence under the other semantics, that means any observed execution on a real implementation (for which the relaxed semantics are a model) is equivalent to a legal execution under the simplifying assumption that all stream operations remain in program order.

In this section we have so far discussed a single pair of language definitions, but in fact this chapter presents two. The first pair assumes that the number of values that can be buffered in a stream is unbounded. Unbounded streams are not representative of most real implementations, but they do simplify the equivalence proofs, and serve as a useful stepping stone to the more complicated bounded stream case.

Bounded stream buffers complicate reasoning about streaming programs because threads can be blocked in two different ways: either an input stream can be empty or an output stream can be full. With unbounded stream buffers, empty input streams are the only

[2] "Later" according to a program-order interpretation of the program.

source of thread blocking.

In the bounded stream buffer context it is not only bad cyclic communication patterns that can cause inconsistencies between program-order and relaxed semantics. Communication patterns that involve branching and reconvergence can also be problematic. For example, if thread t_1 sends to both t_2 and t_3, which both send to thread t_4, there are no directed cycles in the communication graph. However, stream operation reordering in t_4 can cause a deadlock that would not exist under the program-order semantics by causing the stream buffers along one of the paths to fill up, which can cause t_1 to block and prevent it from sending along the other path.

In the bounded stream buffer context, programs that have no cyclic or reconvergent communication patterns (in other words, programs whose communication graphs are trees) are guaranteed to behave equivalently under the program-order and relaxed semantics. However, many useful programs are not in this set. One of the important areas for future work in semantics for C-like streaming languages is finding broader classes of programs for which equivalence can be proved. One direction that we believe could be fruitful is programs that send to and receive from different streams at restricted (static) rates.

The proofs in this chapter are all based on dynamic properties of programs: "if a program ever does X during its execution, then Y". We do not address the related issue of predicting whether a program is well-behaved based only on its source code. This static analysis problem is another important direction in which this work could be extended in the future.

Finally, we offer an analysis of one technique the programmer can use to make kernels safer at the expense of optimization opportunities. The common stream primitives in Macah programs are blocking sends and receives; abstractly, a thread will not continue executing past a blocking operation until it has completed. Macah also has non-blocking sends and receives that might fail, but will not stall the thread, and return a Boolean flag indicating success or failure. Polling stream operations behave like blocking operations, but are constructed by putting non-blocking operations in an explicit loop that re-executes the operation until it succeeds. The looping control flow of the polling operations prevent stream operation reordering by making the execution of later operations explicitly dependent upon the success of the polling operation. The fact that polling stream operations

inhibit operation reordering is bad for optimization flexibility, but it does make it easier to prove equivalence of the program-order and relaxed interpretations of a given program.

The broadest classes of programs for which we prove the equivalence of the program-order and relaxed semantics are those that do not have directed cycles (in the unbounded buffer case) or undirected cycles (in the bounded buffer case) in the stream communication graph. For the purpose of these definitions, there is no edge between two threads in the stream communication graph if they only use *non-blocking* stream operations to communicate with each other. Because polling operations are built from non-blocking operations, they do not induce edges in the stream communication graph either.

7.2 Basics

The semantics presented in this chapter are not for full Macah, but rather a small core language with threads, streams and kernels. Using the smaller language allows the semantics and proofs to be more compact and readable, and still presents the same stream operation reordering issues. Core Macah is essentially the call-by-value lambda calculus with some additional expressions to operate on streams.

The only values in core Macah are "42", which is a stand-in for all primitive values, $\lambda x.e$ which is a function definition and x, which is a variable use.

In core Macah the blocking receive expression does include a specific target location for the received value, because in the usual lambda calculus style it simply evaluates to the value received from the stream. The non-blocking send and receive operators in full Macah use Boolean flags that we usually refer to as the "worked" expressions to indicate whether the operation succeeded or not. In core Macah we use a style similar to Church encoding for non-blocking sends and receives. Each non-blocking stream has two "continuation expressions" at the end; if the operation is successful the program evaluates to the first, if it fails the program evaluates to the second. The first continuation expression for the non-blocking receive (e_1) is assumed to be a function that is called with the value that gets received.

Expressions nested inside a kernel expression are evaluated in *kernel mode*, which can change the relative ordering of sends and receives on different streams in the relaxed semantics.

Core Macah	Name	Full Macah Analogue
$(e_0\ e_1)$	Function call	All non-trivial control flow
$\lhd?\ s$	Blocking receive	`lexp <? stream`
$s \lhd!\ e_v$	Blocking send	`stream <! rexp`
$\lessdot?\ s\ e_w\ e_f$	Non-blocking receive	`wexp :: lexp <? stream`
$s \lessdot!\ e_v\ e_w\ e_f$	Non-blocking send	`wexp :: stream <! rexp`
kernel e	Kernel block	`kernel {...}`
v	Values	

Figure 7.2: All of the expressions in the core Macah grammar and their intuitive connections to full Macah statements/expressions.

For convenience we will use expressions like if/then/else, while and recursive function definitions. These can all be translated into lambda expressions using the conventional encodings. We also define polling sends and receives in terms of their non-blocking cousins. Polling stream operations are almost identical to blocking operations. The difference is that the polling versions use the non-blocking variant and explicit looping. Under the program-order semantics there really is no meaningful difference between polling and blocking. Under the relaxed semantics there is a difference that we will describe in more detail later.

$$\texttt{e}\ \texttt{<@?}\ s \doteq \texttt{w := false; do \{ w :: e <? s \} while (!w)}$$
$$\texttt{s}\ \texttt{<@!}\ e \doteq \texttt{w := false; do \{ w :: s <! e \} while (!w)}$$

Or, in core Macah terms ...

$$\trianglelefteq?\ s \doteq \mathsf{let}\ loop\ \_ = \lessdot?\ s\ (\lambda x.x)\ (loop\ 42)\ \mathsf{in}\ (f\ 42)$$
$$s\ \trianglelefteq!\ e \doteq \mathsf{let}\ v = e\ \mathsf{in}\ \mathsf{let}\ f\ \_ = s \lessdot!\ v\ v\ (f\ 42)\ \mathsf{in}(f\ 42)$$

A program *configuration* or *state* is a tuple (S, T) where S is a stream buffer that maps stream ids to stream configurations, and T is a thread system that maps thread ids

to thread configurations. A stream configuration is a tuple with an unbounded positive integer-indexed array of values and some "pointers", the details of which depend on which semantics we are considering. A thread configuration is a tuple that includes an expression evaluation context and a kernel depth counter. A complete program also includes a stream connectivity description that specifies which thread is allowed to send to a stream and which thread is allowed to receive from a stream. Streams are point-to-point; they have exactly one sender thread and one receiver thread.

7.2.1 Non-blocking stream operations

As we will define formally later, non-blocking operations are "weak" in the sense that it is *always* legal for a non-blocking operation to fail, even if it seems like it should succeed. For the purposes of performance analysis, programmers can expect implementations to provide reasonable best effort in terms of non-blocking stream operations succeeding when possible. The motivation for using this weak definition is similar to the situation with `trylock` in C++ [BA08]. Using the strong definition enables some strange "backwards" programming idioms and badly complicates the semantics equivalence proofs.

Here is a very simple example of the difference between weak and strong definitions of non-blocking operations.

t_1		t_2
$\mathtt{s}_2$ <! 42;	$\|$	$\mathtt{temp}_1$ :: y <? $\mathtt{s}_1$;
$\mathtt{s}_1$ <! 43		$\mathtt{temp}_2$ :: z <? $\mathtt{s}_2$

Using strong non-blocking receives, there are three possible results from running this program:

Result	Schedule
$y = 44, z = 45$	t_2 goes first; both receives fail
$y = 42, z = 43$	t_1 goes first; both receives succeed
$y = 44, z = 43$	t_2 starts, t_1 interrupts it, then t_2 finishes; the first receive fails, but the second succeeds

Using the weak definition of non-blocking receives the fourth and most counterintuitive option is also possible, that the first receive succeeds and the second fails.

Preventing the $y = 42, z = 45$ result from occurring in a real implementation requires all stream operations to stay in order, even when different streams are involved. This is an expensive guarantee for an interconnect network to provide. We could explicitly model an interconnect network that might reorder messages between threads in our semantics, but we can achieve the same effect with considerably less complexity by using the weak definition of non-blocking stream operations.

Here's another example based on a `trylock`-based example in [BA08]. It may seem strange, but it gets closer to the importance of the weak definitions.

```
     t1                          t2
s1 <! 42;          ||   s2 <! 42;
temp <? s2              do {
                           w1 :: x1 <? s2
                           if (w1) s2 <! x1;
                        } while(w1);
                        w2 :: x2 <? s1;
                        assert(w2)
```

This example is actually not entirely legal, since both threads receive from a single stream ($\mathtt{s}_2$), but we will pretend that is legal for the purpose of this example. Thread t_2 sends a single value on stream $\mathtt{s}_2$ and then goes into a loop where it is essentially waiting for t_1 to receive it. Since the send to $\mathtt{s}_1$ happens before the receive from $\mathtt{s}_2$, with a strong definition of non-blocking receives, the receive at the end of t_2 will never fail. In a sense t_2 is getting information (whether t_1 has performed the receive or not) *backwards* along $\mathtt{s}_2$, which is a feature that we do not want to support.

Pre-action state	Act.	Post-action state	Cond.
$S, (k,E[\lambda x.e\ v])$	β	$S, (k,E[\{x \mapsto v\}e])$	
$S, (k,E[\mathsf{kernel}\ e])$	kb_{k+1}	$S, (k+1,E[\mathsf{kmode}\ e])$	
$S, (k,E[\mathsf{kmode}\ v])$	ke_k	$S, (k-1,E[v])$	
$S[s \mapsto (r,w,V)], (k,E[\lhd?\ s])$	r_s	$S[s \mapsto (r+1,w,V)], (k,E[V(r)])$	$r < w$
$S, (k,E[\lessdot?\ s\ e_1\ e_2])$	fr_s	$S, (k,E[e_2])$	
$S[s \mapsto (r,w,V)], (k,E[\lessdot?\ s\ e_1\ e_2])$	r_s	$S[s \mapsto (r+1,w,V)], (k,E[e_1\ V(r)])$	$r < w$
$S[s \mapsto (r,w,V)], (k,E[s \lhd!\ v])$	s_s	$S[s \mapsto (r,w+1,V[w \mapsto v])], (k,E[v])$	
$S, (k,E[s \lessdot!\ v\ e_2\ e_3])$	fs_s	$S, (k,E[e_3])$	
$S[s \mapsto (r,w,V)], (k,E[s \lessdot!\ v\ e_2\ e_3])$	s_s	$S[s \mapsto (r,w+1,V[w \mapsto v])], (k,E[e_2])$	

Figure 7.3: Program order semantics with unbounded stream buffers.

7.3 Unbounded stream buffer semantics

Here we present program-order and relaxed semantics with the assumption that stream buffers are unbounded. Bounded streams are more realistic from an implementation perspective, but starting with unbounded streams lets us prove the equivalence of the program-order and relaxed semantics in a simpler setting. In the next section we will introduce stream bounds and discuss the complexities that they add.

Figure 7.3 shows the evaluation rules for the program-order semantics. For simplicity of presentation, each row in the table shows only the state of the global stream buffer and a single thread. A complete evaluation step has to select some thread nondeterministically and apply one of these rules. All other threads are not affected by some thread taking an evaluation step.

Each evaluation rule is labeled with an "action name" (Act. in Figure 7.3). These actions are used in the equivalence proofs. A complete list of the actions is shown in Figure 7.4.

All stream operations in all the semantics presented in this chapter have the additional implicit condition that the thread performing the operation must be the correct one according to the connectivity description for the program. We write "thread t is the sender to

Act.	Meaning
β	Beta-reduction; no externally observable effect
kb_k	Begin kernel level k
ke_k	End kernel level k
r_s	Receive from stream s
s_s	Send to stream s
fr_s	Failed non-blocking receive from s
fs_s	Failed non-blocking send to s
br_s	Receive from s becomes blocked
ur_s	Receive from s becomes unblocked
bs_s	Send to s becomes blocked
us_s	Send to s becomes unblocked
lg_s	Move a value from the local to global buffer for stream s

Figure 7.4: Each evaluation step in the semantics is labeled with an action name.

stream s" $t \hookrightarrow s$, and "thread t is the receiver from stream s" $s \hookrightarrow t$.

kmode is a runtime expression that does not exist in the programmer-visible syntax of the language. Its purpose is to keep track of whether some expression is being evaluated in kernel mode or not. Because it is legal to nest kernels, we add a counter (k) to the state of a thread. This counter indicates how deeply nested in kernel expressions a thread is currently. In the program-order semantics there is no difference between evaluation in kernel mode versus non-kernel mode.

We are mostly concerned with programs that deadlock under one semantics, but not another. Clearly the language we have defined permits deadlocking programs, but we are mostly going to focus on programs that do not deadlock under the unbounded buffers, program-order semantics. Programs that deadlock under even those semantics are probably buggy programs. Some languages like StreamIt make it easy to prove the complete absence of deadlocks [PD10]. However, languages with such guarantees make it hard to program some reasonable stream communication patterns. In analogy with the shared memory

multithreading world, the Macah ecosystem probably needs analysis tools like data race detectors to find programs that might deadlock no matter which semantics is chosen.

7.3.1 Relaxed semantics with unbounded streams

The relaxed semantics are intended to model static reordering of code inside kernels. Instead of syntactically reordering expressions, which would introduce a huge amount of complexity to the semantics and proofs, we chose to model static code reordering as local buffering of stream operations. The local evaluation of each thread is essentially identical under the program-order and relaxed semantics. The difference is that different threads can have different understandings of the order in which stream operations happen.

There are several important things to note about the relaxed semantics in Figure 7.5. The stream configurations have an additional component (b_w): a write buffer count. In kernel mode (and only in kernel mode) when a thread sends a value to a stream, the value does not immediately go to the global stream buffer, but lives in a local buffer for some amount of time. We model this local buffering in the semantics with the buffer count. Sends in kernel mode do not affect the write pointer, but do increment the buffer count. Later a thread can take a step with the last rule that moves a value from the local buffer to the global buffer by incrementing the write pointer and decrementing the buffer count.

If a thread's next expression to evaluate is a receive, but there is no data available from the stream, the thread can take a step into a special blocked state (represented by $\mathfrak{B}$). When a thread is in the blocked state, it cannot send values from its local buffer. This additional restriction has the effect that there are some programs that have deadlocking traces under the relaxed semantics, but not under the program-order semantics.

There is an additional condition on the step that leaves kernel mode (i.e. when k goes from 1 to 0). The condition says that all streams for which the thread making the step is the sender must have no buffered sends. In code reordering terms, this means that the compiler is free to ignore the possibility of dependencies through streams only when it is reordering code inside kernels.

Observation. As long as a thread is in kernel mode some values might be "trapped"

Pre-action state	Act.	Post-action state	Condition
$S, (k, E[\lambda x.e\ v])$	β	$S, (k, E[\{x \mapsto v\}e])$	
$S, (k, E[\mathsf{kernel}\ e])$	kb_{k+1}	$S, (k+1, E[\mathsf{kmode}\ e])$	
$S, (k, E[\mathsf{kmode}\ v])$	ke_k	$S, (k-1, E[v])$	$k > 1$
$S, (1, E[\mathsf{kmode}\ v])$	ke_1	$S, (0, E[v])$	$\forall s$ such that $t \hookrightarrow s$, $S[s \mapsto (r, w, 0, V)]$
$S[s \mapsto (r, w, b_w, V)]$, $(k, E[\lhd?\ s])$	r_s	$S[s \mapsto (r+1, w, b_w, V)]$, $(k, E[V(r)])$	$r < w$
$S, (k, E[\lessdot?\ s\ e_1\ e_2])$	fr_s	$S, (k, E[e_2])$	
$S[s \mapsto (r, w, b_w, V)]$, $(k, E[\lessdot?\ s\ e_1\ e_2])$	r_s	$S[s \mapsto (r+1, w, b_w, V)]$, $(k, E[e_1\ V(r)])$	$r < w$
$S[s \mapsto (r, w, b_w, V)]$, $(k, E[\lhd?\ s])$	br_s	$S, (\mathfrak{B}, k, E[\blacktriangleleft?\ s])$	$r \not< w$
$S[s \mapsto (r, w, b_w, V)]$, $(\mathfrak{B}, k, E[\blacktriangleleft?\ s])$	ur_s	$S, (k, E[\lhd?\ s])$	$r < w$
$S[s \mapsto (r, w, 0, V)]$, $(0, E[s \lhd!\ v])$	s_s	$S[s \mapsto (r, w+1, 0, V[w \mapsto v])]$, $(0, E[v])$	
$S[s \mapsto (r, w, b_w, V)]$, $(k, E[s \lhd!\ v])$	sd_s	$S[s \mapsto (r, w, b_w+1, V[w+b_w \mapsto v])]$, $(k, E[v])$	$k > 0$
$S, (k, E[s \lessdot!\ v\ e_2\ e_3])$	fs_s	$S, (k, E[e_3])$	
$S[s \mapsto (r, w, 0, V)]$, $(0, E[s \lessdot!\ v\ e_2\ e_3])$	s_s	$S[s \mapsto (r, w+1, 0, V[w \mapsto v])]$, $(0, E[e_2])$	
$S[s \mapsto (r, w, b_w, V)]$, $(k, E[s \lessdot!\ v\ e_2\ e_3])$	sd_s	$S[s \mapsto (r, w, b_w+1, V[w+b_w \mapsto v])]$, $(k, E[e_2])$	$k > 0$
$S[s \mapsto (r, w, b_w, V)]$, (k, E)	lg_s	$S[s \mapsto (r, w+1, b_w-1, V)]$, (k, E)	$b_w > 0$

Figure 7.5: The local send buffering semantics with unbounded streams

in the local buffers of the streams that it sends to. The action for leaving kernel mode under the relaxed semantics requires that the local buffers be empty, so it is not until the sending thread has left kernel mode that we can guarantee that all the values it has sent are "visible".

7.3.2 Simple example of deadlock

Here is a simple program that illustrates the difference between the program-order semantics and the local send buffer semantics.

Initially $S = \{\mathsf{s_1} \mapsto (1, 1, \varepsilon), \mathsf{s_2} \mapsto (1, 1, \varepsilon)\}$.

t_1		t_2
kernel{	$\parallel$	$\mathtt{x_2} <? \mathtt{s_1}$;
$\mathtt{s_1} <! 42$		$\mathtt{s_2} <! \mathtt{x_2}$
$\mathtt{x_1} <? \mathtt{s_2}$		
}		

First we consider how it executes under the program-order semantics. Thread $\mathsf{t_2}$ is stuck initially because it needs to receive on stream $\mathsf{s_1}$ and initially the stream is empty. The kernel "wrapper" under the program-order semantics is essentially meaningless, so thread $\mathsf{t_1}$ can evaluate its send along stream $\mathsf{s_1}$. Thread $\mathsf{t_1}$ then becomes stuck on its receive from stream $\mathsf{s_2}$. Thread $\mathsf{t_2}$ can now proceed with its receive from $\mathsf{s_1}$, then do its send to $\mathsf{s_2}$. Finally $\mathsf{t_1}$ can receive from $\mathsf{s_2}$ and now both threads have evaluated to 42. We made no meaningful nondeterministic choices along the way; every execution of this program will have the same result.

Next we look at the execution of this program under the local send buffering semantics. Just like before, $\mathsf{t_2}$ cannot make any progress initially. Under the buffering semantics the send in $\mathsf{t_1}$ goes to the local buffer, because it happens when the thread is in kernel mode. After $\mathsf{t_1}$ sends to the local buffer we arrive at the only important nondeterministic choice in this example: $\mathsf{t_1}$ can send 42 from its local buffer to the global stream buffer or it can attempt to perform a receive from $\mathsf{s_2}$. If $\mathsf{t_1}$ "chooses" the receive we are in trouble, $\mathsf{s_2}$ is empty, so $\mathsf{t_1}$ will go to the blocked receive state and both threads will be stuck. $\mathsf{t_1}$ will

never send to s_1 because of the condition that threads cannot send from their local buffer to the global buffer if their current expression is a blocked receive, and t_2 will never send to s_2 because it is waiting to receive from s_1 first.

This example shows that there exist programs that can deadlock under the buffering semantics but cannot under the program-order semantics.

7.3.3 Equivalence of program-order and relaxed

The program-order semantics and relaxed semantics are clearly different for some programs, but they are equivalent for large classes of interesting programs. Specifically, for all programs that do not deadlock under the program-order semantics and are "well-behaved" in ways that explained below, the semantics are equivalent.

To prove equivalence of the semantics we show that every non-deadlocking trace of actions under the program-order semantics has an equivalent trace under the relaxed semantics and that every trace under the relaxed semantics falls into one of three categories:

- It has an equivalent trace under the program-order semantics.
- Its equivalent trace under the program-order semantics leads to deadlock, in which case the non-deadlocking program assumption has been violated.
- It leads to a deadlocking trace under the relaxed semantics, in which case the program is not "well-behaved".

Lemma 1 (Relaxed simulates program-order) *For every non-deadlocking trace of actions under the program-order semantics there is an equivalent trace under the relaxed semantics.*

Proof by induction on the length of the trace. The invariant is that both program-order and relaxed configurations are exactly equivalent after every program-order action and mirroring relaxed action(s). The base case is simple: the streams are all empty and each thread is just in its initial configuration.

The inductive step is also relatively simple. Most of the actions in the program-order trace can be mirrored one-to-one by equivalent relaxed semantics actions. The most important exception is sending in kernel mode. Under the relaxed semantics a local send followed

immediately by a global send from the same thread to the same stream has the equivalent effect as a send under the program-order semantics and leaves the buffered send count for the stream at zero. There is no need to ever select an action that leads to a blocked state, so we do not need to worry about those. And the additional condition on leaving kernel mode in the relaxed semantics is easily satisfied because we chose to pair every local send immediately with a global send. ⊠

In code reordering terms, Lemma 1 simply shows that it is possible for the compiler and hardware to leave all stream operations in their natural program order, in which case the meaning of the program does not change.

Lemma 2 (program-order simulates relaxed) *For every non-deadlocking trace of actions under the relaxed semantics there is an equivalent trace under the program-order semantics.*

Proof by induction on the length of the relaxed trace. The base case is the same trivial argument as the last proof.

The inductive case requires a more subtle invariant. The write pointers on the streams can get out of sync because a local send under the relaxed semantics does not increment the write pointer whereas a send under the program-order semantics does. The invariant for stream states is that the write pointer in the program order configuration will always be equal to the write pointer plus the buffer count in the relaxed configuration. Also a relaxed thread in blocked mode on a stream op is considered equivalent to a program-order thread that has an equivalent stream op in the evaluation hole.

Showing that each step preserves the invariant is simple for most of the actions. Blocking and unblocking a thread do not change any state relevant to the invariant. Receiving, evaluating a function call, entering kernel mode, sending outside of kernel mode and failed non-blocking stream operations are all trivial.

Sending in kernel mode is more interesting, because the write pointer does not change in the relaxed semantics, but it does in the program-order semantics. The buffered send count does change though, which maintains the invariant. The global send action also maintains the invariant because it increments the write pointer and decrements the buffered send count.

The global send action has the condition that the buffered send count not go negative, so it is impossible for the write pointer in the relaxed configuration to get higher than the write pointer in the program-order configuration. ⊠

Observation. The equivalence relation between program-order configurations and relaxed configurations is one-to-many. What the above lemmas *do not* prove is that one can choose an arbitrary pair of related configurations and proceed happily from there. For example, consider this example.

```
    t1                t2                t3
s1 <! 42   ||  kernel{        ||  x1 <? s2
                  s2 <! 43
                  x2 <? s1
               }
```

The interesting relaxed trace is the one where $t2$ enters kernel mode, locally sends to $\mathtt{s}_2$, then blocks. The equivalent program-order trace enters kernel mode, then performs a normal send to $\mathtt{s}_2$. From that program-order configuration we can choose to either send in t_1 or receive in t_3, whereas in the relaxed configuration sending in t_1 is the only option, because the value in $\mathtt{s}_2$ is "trapped" in t_2's local buffer.

The reason that this apparent disparity between the semantics is not a problem is that we only need to prove that for each possible program-order trace there exists some valid relaxed trace, and vice-versa. It is not necessary to prove that from every possible pair of related intermediate states we can take equivalent steps under both semantics.[3]

Lemma 3 (Deadlocking under the relaxed semantics) *Every relaxed semantics trace that leads to a deadlocked configuration either leads to a deadlocked configuration under the program-order semantics or leads to a relaxed configuration in which there is a cycle in the thread and stream graph in which every thread is stalled waiting on a receive from its predecessor. Also, at least one thread in the cycle must be blocked in kernel mode and have locally buffered values for its successor in the cycle.*

---

[3]Which is lucky, because it is not true.

Proof by case analysis of the possible deadlocked configurations. There are many programs that can deadlock under both the program-order semantics and relaxed semantics; these are not interesting programs because it is usually considered desirable to prove (formally or informally) that programs are deadlock-free under the program-order semantics.

The more interesting case is traces under the relaxed semantics that lead to a deadlock where the equivalent program-order trace can still make progress. There are two interesting sub-cases to consider:

- Exiting kernel mode. A thread cannot exit kernel mode if there are buffered values on any of the streams it sends to. However, if both the relaxed and program-order traces manage to reach a configuration where $\mathsf{kmode}\ v$ is the next expression to evaluate, the program under the relaxed semantics is not actually deadlocked, because the global sending action can be applied.
- Under the unbounded stream buffer semantics, the only kind of expression a thread can get deadlocked on is a receive. So there must be at least one thread that can receive under the program-order semantics, but cannot under the relaxed semantics because the value in question is stuck in the sender's local buffer. The sender must also be deadlocked on a receive, or else the program would be able to make progress. We can continue tracing back this way until we find a cycle in the sequence of blocked threads. It is impossible for there not to be a cycle, because that would require some "source" thread that is deadlocked but is not waiting for a receive from some other thread. But there is no way to deadlock without blocking on a receive.

  Finally, at least one of the streams must have values trapped in the local buffer of its sender. Otherwise the relaxed and program-order configurations of these streams would be the same and we have found a deadlocked configuration under the program-order semantics, which we assumed was not the case.

⊠

Theorem 4 (Unbounded stream equivalence) *Every program that cannot deadlock under the program-order semantics either (a) behaves equivalently under program-order and*

relaxed semantics or (b) might reach a deadlocked configuration under the relaxed semantics with at least one thread that is blocked on a receive and has unsent buffered values.

Proof by a straightforward combination of Lemmas 1, 2 and 3. ⊠

Theorem 4 gives the fundamental symptom shared by all programs that differ under the program-order and relaxed semantics, but does not give much intuition about what kinds of programs these are. An example of a class of programs that trivially cannot exhibit relaxed semantics deadlocking is those that have no receives at all. A more interesting class is captured in the following theorem.

Theorem 5 (Non-blocking receive in cycles) *Every program that cannot deadlock under the program-order semantics and uses only non-blocking receives in kernel mode on streams that participate in some cyclic (i.e. feedback) path in the stream graph behaves identically under program-order and relaxed semantics.*

Proof by extension of Theorem 4. Using the previous theorem, we only have to show that not using non-blocking receives in kernel mode on "feedback streams" makes it impossible for a thread to be simultaneously blocked on a receive and have values locally buffered. Only blocking receives can cause a thread to enter the blocked state, so threads with only non-blocking receives will not. ⊠

Note that any program can be adapted to fit the constraints of theorem 5 by replacing blocking receives with polling receives. Polling receives behave almost exactly like blocking receives, but they let buffered values drain out, which prevents the kind of deadlock that the relaxed semantics can introduce. Polling receives are generally much more expensive than blocking receives, which is why the blocking version exists at all.

Observation. From any pair of a relaxed configuration and its equivalent program-order configuration, there are a finite number of legal steps under the relaxed semantics that correspond to no action at all under the program-order semantics (i.e. blocking or unblocks, global sending). This means that under the relaxed semantics progress will be made.

Lingering issue. There are some programs that don't quite fit in the equivalence proof the way it seems they should. For example:

```
      t1                        t2
for (...) {        ||   kernel {
    x1 ⊴? s2                     for (...) {
    s1 ◁! x1                         s2 ◁! x2
}                                    x2 ◁? s1
                                 }
                         }
```

If the receive in t_1 was blocking, this would be a totally standard example of a program that works fine under the program-order semantics, but can deadlock under the relaxed semantics. However, the receive is polling, which means that if t_2 does a local send to s_2, then goes into blocking mode on the receive from s_1, t_1 will just spin in the polling receive forever.

In some sense this is a possible execution under the program-order semantics, but it is not a very interesting one. The execution in which t_2 essentially never executes represents a totally unfair schedule.

There may be some way to extend the proof to handle cases like this. For example, maybe we could say that any trace where some thread cannot make progress after an unbounded number of steps is considered a deadlocked trace.

7.3.4 More complex communication patterns

Theorem 5 proves that the equivalence of the program-order and relaxed semantics applies only to some programs. The set of programs covered is a useful one, but excludes many reasonable programs. Here is an example pattern that we found to be fairly common in practice.

```
        t1                          t2
while (...) {          ||   kernel {
    s1 ◁! e1                    while (...) {
    for (...) {                     x2 ⊴? s1
        ...                         for (...) {
        s1 ◁! e2                        x3 ◁? s1
    }                                   ...
    for (...) {                     }
        x1 ◁? s2                    for (...) {
        ...                             ...
    }                                   s2 ◁! e3
}                                   }
                                }
                            }
```

This sketch represents an iterative computation where some part of each iteration (t_2) can be accelerated and some part cannot (t_1). The non-kernel thread sends some input data to the kernel thread, which does some computation on it and sends results back. That whole pattern repeats in the while loop. The program is not covered by Theorem 5, because in kernel mode it performs a blocking receive from a stream that's part of a feedback loop.

Even though it seems like it might, in fact this program cannot deadlock. Informally, the polling receive at the beginning of each outer iteration in t_2 allows all the sends to complete without getting blocked. In compiler scheduling terms, even if the send in t_2 is scheduled very late, the loop that is implicit in polling receives will keep the thread executing and eventually all the sends will execute.

Generalizing from this example to an interestingly broad class of programs for which the program-order and relaxed semantics are provably equivalent would be useful for static analysis tools whose job is to check the safety of programs written in Macah (and similar streaming languages). We believe that a number of such patterns exist; finding and proving equivalence for them is an important piece of future work.

7.4 Blocking and polling

As observed in the previous section, programmers can simply always use polling receives instead of blocking receives to guarantee that their programs behave identically under unbounded program order and relaxed semantics. There are two reasons that this programming style is undesirable. The first is related to the relative weakness of the control resources in accelerators. When data is not available for a blocking receive, it is legal for a hardware implementation of a Macah kernel to stall the whole kernel until data is available. This is a relatively inexpensive feature to implement.

Recall that polling receives are just non-blocking receives inside a loop that spins until the receive succeeds. In implementation terms, adding an extra loop where there were just blocking stream operations before significantly complicates the control flow of the kernel.

The second reason that using polling receives is undesirable has to do with the scheduling flexibility given to the compiler. Consider this example that is like our earliest pipelining example, except it uses polling instead of blocking stream operations.

```
for (i = 0..10) {
    x ⊴? s1
    s2 ⊴! x
}
```

Desugaring the polling operations (and using more familiar do/while syntax) gives us:

```
for (i = 0..10) {
    do w :: x ⋖? s1
    while ¬w
    do w :: s2 ⋖! x
    while ¬w
}
```

The "::" notation for non-blocking operations is an alternative syntax where the operation returns a Boolean flag that indicates success or failure, instead of evaluating one expression or another. Because the polling loops will run until the stream operation actually succeeds, later stream operations become control dependent on earlier polling operations and cannot be reordered before them.

This analysis provides an interesting new perspective on blocking operations. The relaxed semantics of Macah essentially say that a program (running in kernel mode) can continue executing operations that come after a blocking stream operation, even if the stream operation has not completed yet. Each thread just assumes that its stream operations will complete eventually, no matter what it goes on to do.

Using polling operations makes explicit the sequencing constraints that are implicit with blocking operations. This has the effect of preventing pipelining and other useful optimizations, which can have a dramatic negative effect on performance.

7.5 Bounded stream buffers

Most streaming languages impose bounds on stream buffer capacities, because unbounded stream buffers mean that the system has to be responsible for automatically allocating and deallocating memory. For accelerators, bounded stream buffers also fit naturally with the bounded local memory resources. Unfortunately, going from unbounded to bounded stream buffers introduces significant complexity to stream operation reordering. The main addition to the formal semantics is a capacity function ($\mathfrak{C}$) that maps stream IDs to the maximum number of values that stream can hold at any given time. $\mathfrak{C}$ does not change during the execution of a program. The formal semantics with program-order stream operations and bounded buffers is given in Figure 7.6.

To demonstrate the difference between unbounded and bounded semantics (independent of program-order versus relaxed issues), here is a simple program that cannot deadlock with unbounded streams, but will deadlock if the capacity of s_1 is 1:

$$\begin{array}{cccc} & t_1 & & t_2 \\ \mathsf{s}_1 \lhd! \, 42; \mathsf{s}_1 \lhd! \, 43; \mathsf{s}_2 \lhd! \, 44 & \| & \lhd? \, \mathsf{s}_2; \lhd? \, \mathsf{s}_1; \lhd? \, \mathsf{s}_1 \end{array}$$

t_1 tries to send two values to s_1 before sending anything to s_2, but t_2 tries to receive from s_2 first. t_1 will block on the second send to s_1, because the stream buffer is full, and t_2 will never make any progress.

Pre-action state	Act.	Post-action state	Condition
$S, (k, E[\lambda x.e\ v])$	β	$S, (k, E[\{x \mapsto v\}e])$	
$S, (k, E[\mathsf{kernel}\ e])$	kb_{k+1}	$S, (k+1, E[\mathsf{kmode}\ e])$	
$S, (k, E[\mathsf{kmode}\ v])$	ke_k	$S, (k-1, E[v])$	
$S[s \mapsto (r,w,V)], (k, E[\lhd?\ s])$	r_s	$S[s \mapsto (r+1,w,V)], (k, E[V(r)])$	$r < w$
$S, (k, E[\lessdot?\ s\ e_1\ e_2])$	fr_s	$S, (k, E[e_2])$	
$S[s \mapsto (r,w,V)]$, $(k, E[\lessdot?\ s\ e_1\ e_2])$	r_s	$S[s \mapsto (r+1,w,V)], (k, E[e_1\ V(r)])$	$r < w$
$S[s \mapsto (r,w,V)], (k, E[s \lhd!\ v])$	s_s	$S[s \mapsto (r,w+1,V[w \mapsto v])], (k, E[v])$	$w - r \leq \mathfrak{C}_s$
$S, (k, E[s \lessdot!\ v\ e_2\ e_3])$	fs_s	$S, (k, E[e_3])$	
$S[s \mapsto (r,w,V)]$, $(k, E[s \lessdot!\ v\ e_2\ e_3])$	s_s	$S[s \mapsto (r,w+1,V[w \mapsto v])], (k, E[e_2])$	$w - r \leq \mathfrak{C}_s$

Figure 7.6: Program order semantics with bounded buffers

7.5.1 Relaxed semantics with bounded-capacity streams

In the bounded stream context, the intuition for what the relaxed semantics are intended to model is the same. Inside kernels, the compiler should be allowed to reorder stream operations in any way that respects local dependencies and keeps all operations for a particular stream in order.

There is an important choice to make in the relaxed semantics: should the capacity restriction be enforced on the local send action or the global send action? The local send seems natural, because the number of values in the local and global buffers together would be limited by the capacity of the stream. However, given that the idea is to model compiler reorderings, it seems that enforcing the capacity restriction on the global send action is the only option; otherwise the system would be unnaturally restricted in the ways it could schedule a kernel. The formal semantics for the relaxed, bounded buffers case is shown in Figures 7.7 and 7.8.

Here is an example that is guaranteed to not deadlock under both the program-order

Pre-action state	Act.	Post-action state	Condition
$S, (k, E[\lambda x.e\ v])$	β	$S, (k, E[\{x \mapsto v\}e])$	
$S, (k, E[\mathsf{kernel}\ e])$	kb_{k+1}	$S, (k+1, E[\mathsf{kmode}\ e])$	
$S, (k, E[\mathsf{kmode}\ v])$	ke_k	$S, (k-1, E[v])$	$k > 1$
$S, (1, E[\mathsf{kmode}\ v])$	ke_1	$S, (0, E[v])$	$\forall s$ such that $t \hookrightarrow s$, $S[s \mapsto (r, w, 0, V)]$
$S[s \mapsto (r, w, b_w, V)]$, $(k, E[\lhd?\ s])$	r_s	$S[s \mapsto (r+1, w, b_w, V)]$, $(k, E[V(r)])$	$r < w$
$S, (k, E[\lessdot?\ s\ e_1\ e_2])$	fr_s	$S, (k, E[e_2])$	
$S[s \mapsto (r, w, b_w, V)]$, $(k, E[\lessdot?\ s\ e_1\ e_2])$	r_s	$S[s \mapsto (r+1, w, b_w, V)]$, $(k, E[e_1\ V(r)])$	$r < w$
$S[s \mapsto (r, w, b_w, V)]$, $(k, E[\lhd?\ s])$	br_s	$S, (\mathfrak{B}, k, E[\blacktriangleleft?\ s])$	$r \not< w$
$S[s \mapsto (r, w, b_w, V)]$, $(\mathfrak{B}, k, E[\blacktriangleleft?\ s])$	ur_s	$S, (k, E[\lhd?\ s])$	$r < w$

Figure 7.7: The local send buffering semantics with bounded streams and unbounded reordering (Part 1/2).

semantics with bounded buffers and the relaxed semantics with unbounded streams, but might deadlock under the relaxed semantics with bounded streams.

$$\underset{t_1}{\mathsf{kernel}\ (\mathsf{s}_1 \lhd!\ 42; \mathsf{s}_2 \lhd!\ 43; \mathsf{s}_2 \lhd!\ 44)} \quad \| \quad \underset{t_2}{\lhd?\ \mathsf{s}_1; \lhd?\ \mathsf{s}_2; \lhd?\ \mathsf{s}_2}$$

The problematic trace is when t_1 performs a local send of 42 and 43, then tries to send 44 to s_2. Because s_2's capacity is 1 and there is already a value buffered in it, t_1 will block. Because t_1 is blocked in kernel mode on a send to s_2, it will never release the values in its local buffers, so t_2 can never receive from s_1, and the system is deadlocked.

Here is another example that shows just how problematic the combination of relaxed

Pre-action state	Act	Post-action state	Condition
$S[s{\mapsto}(r,w,0,V)]$, $(0,E[s \lhd! v])$	s_s	$S[s{\mapsto}(r,w{+}1,0,V[w{\mapsto}v])]$, $(0,E[v])$	$w{-}r \leq \mathfrak{C}_s$
$S[s{\mapsto}(r,w,b_w,V)]$, $(k,E[s \lhd! v])$	sd_s	$S[s{\mapsto}(r,w,b_w{+}1,V[w{+}b_w{\mapsto}v])]$, $(k,E[v])$	$k > 0$
$S, (k,E[s \lessdot! v\ e_2\ e_3])$	fs_s	$S, (k,E[e_3])$	
$S[s{\mapsto}(r,w,0,V)]$, $(0,E[s \lessdot! v\ e_2\ e_3])$	s_s	$S[s{\mapsto}(r,w{+}1,0,V[w{\mapsto}v])]$, $(0,E[e_2])$	$w{-}r \leq \mathfrak{C}_s$
$S[s{\mapsto}(r,w,b_w,V)]$, $(k,E[s \lessdot! v\ e_2\ e_3])$	sd_s	$S[s{\mapsto}(r,w,b_w{+}1,V[w{+}b_w{\mapsto}v])]$, $(k,E[e_2])$	$k > 0$
$S[s{\mapsto}(r,w,b_w,V)], (k,E)$	lg_s	$S[s{\mapsto}(r,w{+}1,b_w{-}1,V)], (k,E)$	$b_w{>}0 \wedge w{-}r{\leq}\mathfrak{C}_s$
$S[s{\mapsto}(r,w,b_w,V)], (k,E)$	bs_s	$S, (\mathfrak{B}_s,k,E)$	$b_w{>}0 \wedge w{-}r{\not\leq}\mathfrak{C}_s$
$S[s{\mapsto}(r,w,b_w,V)]$, $(\mathfrak{B}_s,k,E)$	us_s	$S, (k,E)$	$w{-}r \leq \mathfrak{C}_s$

Figure 7.8: The local send buffering semantics with bounded streams and unbounded reordering (Part 2/2).

kernels and bounded buffers is

```
        t1                    t2                    t3
x1 := 0.1             ||  kernel {          ||  x3 ◁? s2
while (x1 < 0.5) {            s2 ◁! 0.1           while (x3 < 0.5) {
    x1 := random()            x2 ◁? s1                x3 ◁? s3
    s3 ◁! x1              }                       }
}
s1 ◁! 42
```

The problem in this program is that t_1 sends to t_2 after its big loop and t_3 receives from t_2 before its big loop. Under the program-order semantics, this program will work fine. First t_2 can send to t_3. Then t_1 and t_3 can communicate as many values as they like

over a stream with a small capacity. Finally t_1 can send a value to t_2.

Under relaxed semantics with unbounded buffers this program is still okay. t_2 might reorder its stream operations (i.e. the send might get buffered locally until after the receive). Now t_3 cannot make progress, because it hasn't received from t_2 yet. But t_1 can send as many value as it likes on s_3. After the loop in t_1 finishes, it will send a value to t_2, which will unblock and send a value on to t_3, which will then be free to receive all the values sent by t_1.

This example shows that even a very small reordering in a thread (t_2) can cause an unbounded amount of capacity to be required on a stream (s_3), that it is not directly connected to.

Here is another interesting example that shows that some programs that deadlock under the bounded program-order semantics might not deadlock under the bounded relaxed semantics with unconstrained reordering. It is essentially exactly the same as the first example in this section, except t_1's code is in a kernel block in this version.

```
     t1                t2
kernel {          ||   ◁? s2
    s1 ◁! 42           ◁? s1
    s1 ◁! 43           ◁? s1
    s2 ◁! 44
}
```

From the compiler reordering perspective, it is clear that the send to s_2 can be reordered before the other sends, and the example works just fine with buffer capacities of 1. Using the relaxed semantics, the first two sends can be stored in the local buffer while the last send executes.

This example shows that under the relaxed semantics a program might "get lucky" and execute in a way that it appears there is more buffering capacity than is specified by the program.

7.5.2 Equivalence with Bounded Buffers

As the examples in the previous section suggest, proving equivalence between program-order and relaxed semantics with bounded buffers requires very careful definition of "equivalence" and exactly what set of programs it applies to.

Lemma 6 (*Unbounded* program-order simulates bounded relaxed) *For every non-deadlocking trace of actions under the bounded relaxed semantics there is an equivalent trace under the unbounded program-order semantics.*

Proof by induction on the length of the trace. This proof is essentially the same as the proof that the unbounded program-order semantics simulate the *unbounded* relaxed semantics (Lemma 2). The bounded relaxed semantics can hit deadlocking situations that would not deadlock under the unbounded semantics, but these traces are not relevant to this proof. ⊠

The importance of this lemma is that in the bounded buffers case there are non-deadlocking traces under the relaxed semantics that cannot be simulated by the program-order semantics. This is exactly because of the "lucky" reorderings referred to above. However, with this lemma we can say that all such executions would have been possible if the stream buffers were unbounded. In other words, under the relaxed semantics the system is allowed to capriciously behave as if some buffers have more capacity than specified.

Lemma 7 (Bounded relaxed simulates bounded program-order) *For every non-deadlocking trace of actions under the bounded program-order semantics there is an equivalent trace under the bounded relaxed semantics.*

Proof by induction on the length of the trace. This proof is essentially identical to the proof of Lemma 1, which states that the unbounded relaxed semantics simulate the unbounded program-order semantics. ⊠

Lemma 8 (Deadlocking with bounded streams) *Every trace under the bounded relaxed semantics that leads to a deadlocked configuration either leads to a deadlocked configuration under the program-order semantics or leads to a relaxed configuration in which there is an* un*directed cycle in the stream graph in which every thread is stalled waiting on*

a receive or send from a neighbor. Also, at least one thread in the cycle must be blocked in kernel mode and have locally buffered values for its successor in the cycle.

Proof idea by case analysis of the possible deadlocked configurations. This proof is similar to the proof of Lemma 3 for the unbounded semantics. There are more cases, because threads can block either sending or receiving, but the high level concept is the same: There must be some thread that has buffered sends and is in an undirected cycle of threads that are stuck waiting for those values to come out.

In the unbounded case we showed that using polling receives on any "feedback streams" guarantees that the program-order and relxed semantics are equivalent. The corresponding programming pattern in the bounded case is to use polling receives *and* sends on all feedback streams and "reconvergent fanout streams". Reconvergent fanout streams are those that participate in undirected cycles in the stream graph, but not directed cycles. Such a severe restriction would likely have a substantial negative impact on the performance of many programs.

7.6 Future work

7.6.1 Less severe restrictions

Finding useful classes of programs that can be proved equivalent with bounded buffers is an important direction for future work. For example, even though Macah allows free use of conditional and non-blocking stream operations, programs could adhere to a more limited style, like synchronous dataflow. In synchronous dataflow, threads perform sends and receives in some statically known simple pattern.

The trouble with synchronous dataflow is that it is too restrictive for some applications that have data-dependent behavior. To use the shared memory analogy again, programming strictly with critical sections and no data races works well most of the time, but for some applications it is important to have other mechanisms like atomic read-modify-write primitives. Perhaps there are equivalent techniques that could be used to write streaming programs that are both safe and efficient.

7.6.2 Receive reordering

The relaxed semantics with send buffers are actually more strict than our current implementation. The semantics dictate that all receives happen in program order, whereas our implementation allows receives from different streams to be reordered.

A simple program that cannot deadlock under any of the semantics we have looked at so far, but can deadlock if we allow receive reordering (the capacity of s_1 is 1):

```
   t1                t2
s1 <|! 42   ||   kernel {
s1 <|! 43            x1 <|? s1
s2 <|! 44            x2 <|? s1
                     x3 <|? s2
                 }
```

In the deadlocking trace, t_1 sends 42 to s_1. Then t_1 blocks on the next send, because the capacity of s_1 is 1. t_2 can demand to receive on s_2 first because we are allowing receive reordering, and if it does the program is now deadlocked. t_1 "wants" to send to s_1 and t_2 "wants" to receive from s_2.

Formalizing receive reordering seems more complicated than send reordering. For send reordering, we introduced local buffers that make sends "happen late" from the perspective of the global stream buffer. It is not obvious how to make receives "happen early" in an analogous way.

One possibility is to replace the buffering in the relaxed semantics proposed in this chapter with a form of lazy evaluation. The main idea is that when a thread evaluates a receive or send in kernel mode it does not actually perform the operation, but just records the fact that it has to do it at some point in the future. The thread can then continue executing, lazily delaying anything that depends (directly or indirectly) on the operation. Similar to the nondeterministic sending from the local buffer, the thread can choose to evaluate lazily built-up computations whenever it wants to.

This seems like a correct way to define programs, but I am afraid of trying to prove

anything useful with all the extra cruft the lazy evaluation would require.

If I could get the lazy evaluation version working it might subsume the local buffers entirely which seems maybe nice.

7.6.3 Shared memory

Most C-like languages for accelerators do not support direct access to shared memory through pointers inside of kernels.[4] Accelerator hardware is not designed to support random access of memory well, so it makes sense to force programmers to think differently about memory by accessing it through streams.

Shared memory in kernels can be implemented by translating memory accesses into sending an address to a special memory controller thread, and receiving the data back over another stream. The programmer can do this translation by hand today, and it might make sense to provide some automatic support.

7.7 Summary

Many existing C-like languages for accelerators use a combination of kernels that are automatically parallelized by the compiler and streams for inter-thread communication and memory access. We showed that simply allowing the compiler to make reordering transformations to kernels restricted only by local dependencies leads to differences between the (intuitive) program-order semantics and the (actually implemented) relaxed semantics.

We believe that the best way to define a C-like language for accelerators is with both program-order and relaxed semantics. The relaxed semantics represent the true meaning of the language, and the program-order semantics provide a more intuitive model for programmers. We proved that for some interesting classes of programs the program-order and relaxed semantics are equivalent, which means that thinking in terms of the program-order semantics is safe.

Not all useful programs fit the patterns for which we proved equivalence, which means: (a) there is more work needed in proving equivalence for wider sets of useful programs; and

[4] Languages and compilers that do support arbitrary pointer code face the challenge of proving that after optimizations like deep loop pipelining, memory reads and writes appear to execute in the original order.

(b) there is a need for analysis tools that will tell programmers whether their program is definitely safe, definitely dangerous, or too complex to say for sure.

We hope that this work provides motivation for researchers and vendors working on C-like languages for accelerators to define formal semantics for their languages. In this space it is common for languages to be "defined" by a single implementation alone, which is a major obstacle to portability.

Chapter 8

CONCLUSIONS AND FUTURE WORK

Parallel coprocessor accelerators offer large performance and energy efficiency advantages compared to conventional processors on a range of important applications. As the performance scaling of conventional processors has slowed, interest has grown in accelerators as a means to implement ever more powerful computer systems, from scientific computing clusters to media-rich handheld devices. However, the languages currently in wide use for programming accelerators are unfamiliar to most programmers and are generally (and correctly) considered hard to use compared to conventional languages used for performance-sensitive applications, like C/C++ and Fortran.

In this dissertation we explored abstract architectural models, language features, compilation techniques, and formal semantics for making the programming of accelerators easier. Together, these projects represent significant progress in making accelerators an attractive tool for a wider range of programmers. Our specific goal is enabling "C-level" programming, and we have taken clear steps in that direction.

8.1 Summary of results

The HMP model (Chapter 2) that we proposed provides an abstract picture of the hardware resources available for algorithm designers to use when working with accelerators. We demonstrated that using only the model, we can identify ways in which algorithms need to be restructured to take advantage of accelerators. The model was an important tool for teaching undergraduate students about the most important differences between conventional processors and accelerators. Even members of the Mosaic group who are experts in accelerator architectures found it useful to refer back to the model during application development work to stay focused on the most fundamental performance issues.

Enhanced loop flattening (Chapter 4) helps remove unnecessary limitations on the style

of control flow that accelerator programmers can use. We showed that it is possible to pipeline complex loop nests with the same level of efficiency as conventional inner-loop-only pipelining. It is still possible for programmers to write loops that do not perform well because of application-level dependencies. However, we can now compile complex kernels without fear of the system introducing its own layer of inefficiency due to control flow complexity.

Our experiments with auto-tuning applications for accelerators (Chapter 6) demonstrate that hard architectural resource constraints are an important problem, especially when the relationships between application-level tuning parameters and architecture-level constraints are complex. We developed a new auto-tuning search algorithm that uses probabilistic estimates of many program features to predict both the quality and likelihood of success for untested configurations. The success of this search method means that programmers can target families of related accelerators, instead of thinking about the exact capacities of a particular chip.

We identified an important language semantics issue related to the deep pipelining of kernels and the use of streams for communication and memory access. We proposed a relaxed semantics (Chapter 7) that explicitly models reordering of stream send and receive operations, which is provably equivalent for "well-behaved" programs to a semantics under which all stream operations are kept in program-order. These semantics are a basis on which tools to help programmers identify "ill-behaved" parts of their programs could be built.

8.2 How far have we come?

To evaluate the significance of our progress, we take a step back from these specific contributions. Our larger goal is to make programming accelerators as easy as possible, with the critical caveat that there is no point in using accelerators if we give up too much performance for convenience. At one extreme is current practice: hardware description languages for programming FPGAs and CUDA/CTM for programming GPUs. According to the most recent application studies that compare multiple accelerator families [THL09, GBL10, BNW+10], CUDA and CTM have a shallower learning curve than HDLs, but they all require a non-

trivial amount of architecture-level thinking from the programmer.

At the other extreme are compiler research projects that aim to make the existence of accelerators nearly invisible to programmers. If we had effective "accelerating" compilers for conventional languages, accelerators could just be treated as a different kind of processor with slightly different performance characteristics. Unfortunately, many of the technical impediments to automatic acceleration are the same as those that automatically parallelizing compilers have run up against. After several decades of research, automatically parallelizing compilers have seen limited success. In fact, the increasing popularity of self-tuning libraries and general purpose auto-tuning show that even standard processors are now sufficiently complex that fully automatic compilation of unoptimized code to modern processors results in relatively poor performance. The parallelism and hard resource constraints of accelerators make them harder to compile to than even the most complex conventional processors. Given all these challenges, I do not believe that fully automatic compilation to accelerators from conventional languages will be a practical reality for the foreseeable future.

Between these two extremes, C-level programming of accelerators is an attempt to balance the strengths of humans and compilers. We aim to hide as many details of an accelerator as possible, but not the essential differences between accelerators and conventional processors. This is exactly our goal for Macah and the Mosaic toolchain, and to evaluate how much progress we made towards that goal, we will consider three questions:

- Are there applications that should work well on accelerators, but are hard to express in Macah?
- How much extra effort is required from a programmer to go from a good sequential implementation of an application to a high performance Macah implementation?
- How architecture-independent are Macah programs?

8.2.1 Gaps in expressiveness

We have done development work on many applications, and most of them worked well in the Macah/Mosaic framework. One of the important gaps that we identified is the assumption that the logical unit of acceleration in an application is a single kernel. Kernels in Macah

can have a large number of parallel operations, but they are all scheduled in synchrony with each other; Macah does not have kernel with asynchronous sub-parts.

Work has already begun on addressing this issue in the "Mosaic 2" toolchain. In Mosaic 2, applications can have multiple kernels that run asynchronously in different tasks at the same time. This support is critical for applications that have sub-components that run at different data-dependent rates, like (de)compression or data-dependent filtering. Adding multikernel support to Macah/Mosaic introduces some interesting compiler problems, but we believe they can be solved.

A smaller issue that has come up a number of times is that loop flattening/pipelining do a good job of parallelizing primitive operations, but they do not parallelize whole loops with other loops. Other optimizations like loop fusion and skewing are sometimes very profitable, and automatically performing these transformations is not theoretically challenging. However, deciding where and when such transformations should be applied is a bigger challenge. In the spirit of the C-level balance, I believe that C-like languages for accelerators should come with a library of hints that guide the compiler to where it should apply various optimizations. Ideally, the application of these hints would be tunable. By standardizing these hints as part of the language, they can be portable across different compilers.

8.2.2 Application development experience

Close to two dozen programmers, many of them undergraduate students at the University of Washington, have done application development work in Macah. From their experience we can draw some qualitative conclusions about the difficulty of C-level programming of accelerators. With a couple of training sessions and a tutorial that goes through the steps necessary to turn a sequential program into a good Macah implementation, most of the users were able to independently write good Macah code for basic applications like 2D convolution.

The most common source of confusion for Macah programmers is that much of the code looks like conventional sequential C code. This makes it hard for them to visualize the loop parallelization that the compiler performs. This challenge in turn makes it hard to

understand why particular dependencies in a program are obstacles to good performance on an accelerator.

We have also found that for more complex applications, the HMP model gives enough detail to get started on a good accelerator implementation, but often issues come up that require the programmer to "look under the hood", for example, at a more detailed picture of the local memory hierarchy. Learning about this additional layer of architectural detail is an extra burden on the programmer.

In the future work section below, I sketch a performance analysis (profiling) tool that I believe could go a long way towards demystifying the disconnect between sequential-looking code and parallelized implementation. The issue of having to look underneath the model is one for which I do not see an immediate compiler/tool solution. Perhaps a two layer model would be appropriate, where the second layer has a more concrete picture of the accelerator hardware, but still leaves out some details.

8.2.3 How portable is portable?

The Mosaic toolchain was designed to model and compile to a range of coarse-grained reconfigurable architectures, a category that does not include accelerators with a strong SIMD character (GPUs), accelerators with many independent control domains (MPPAs) or fine-grained architectures (FPGAs). We believe that extending the Mosaic toolchain to support FPGAs would require a modest amount of effort, and would not require changing Macah or most programs is a fundamental way.

Adapting Macah and the Mosaic toolchain to massively parallel processor arrays (MPPAs) requires more rethinking of compilation techniques and programming styles. In fact, MPPAs were a major motivation for adding multikernel support to Macah. With that support we expect efficient compilation of Macah to MPPAs to be possible.

Graphics processing units (GPUs) are different from all these other families of accelerators in some important ways. Modern "general purpose" GPUs have hundreds of simple processors that are grouped into blocks of 32 or 64. The blocks are mostly independent of each other, and within each block a single instruction is broadcast to all processors in a

SIMD fashion. GPUs have an important extra degree of flexibility compared to strict SIMD machines: individual processors within a block can branch in different directions. However, only one instruction at a time can be broadcast in a block. This means that if half of the processors branch in one direction and half in another, half of them have to sit idle while the others execute. When the divergent control paths reconverge, all processors can resume executing in parallel.

SIMD execution has a natural connection with data-parallel programming, which is the style that most languages targeted specifically at GPUs use in some way. Macah and the Mosaic toolchain are designed to support a more ad hoc style of parallelism that comes from loop unrolling and pipelining. Whether SIMD-style parallelism and pipeline parallelism can be harmonized in an efficient way is an open research question.

Where does this discussion of different families of accelerators leave our claims about the portability of Macah? The tuning knob search does a good job of adapting a program to particular architectures within a family that differ only in the sizes of various resources. Portability between architectures that differ in more fundamental ways from a single C-level source program is a bigger challenge, and one that we have not yet solved.

At the heart of the cross-family portability challenge is the most imprecise part of the HMP model: the accelerator controller. Different families of architectures have different styles of control that provide different levels of support for three different kinds of parallelism: asynchronous task or thread parallelism, synchronous pipeline parallelism, and symmetric data parallelism.

With the addition of multikernel support in Mosaic 2, Macah will have support for all three styles parallelism, but neither the language nor the model force programmers to use the style that a particular architecture supports best. For the near future I believe that targeting different styles of accelerators will at least require somewhat different programming style, if not different languages. There have been some preliminary efforts to compile languages designed for GPUs to other kinds of accelerators [PGS+09a, PGS+09b, RVDB10], but the results so far have been mixed.

8.3 Promising directions for future work

Throughout this dissertation we have written about possible incremental improvements to the ideas and algorithms we proposed. Here are three bigger-picture ideas for improving the programmability of accelerators.

8.3.1 Healing the GPU/Macah divide

In the previous section we discussed the three different styles of parallel control (task, SIMD, and pipelined) that accelerators can exploit to varying degrees. There is a relatively large gap between Macah and GPUs, because GPUs have strong support for SIMD-style control, and Macah does not force the programmer into a data-parallel style of coding.

One approach to resolving this tension is to change GPUs, rather than the language or compilation approach. The current structure of GPUs is still strongly influenced by their graphics rendering lineage. GPU vendors have only been seriously supporting non-graphics applications on GPUs for a couple of generations. In that time, GPUs have already changed in modest ways to support non-graphics applications (and more complex graphics workloads). Though it was not a commercial success, Intel's Larrabee showed that new accelerators designed for graphics and other workloads do not have to look exactly like today's GPUs. Larrabee had more in common with MPPAs than it did with conventional GPUs from NVIDIA and ATI.

Conventional GPUs already have strong support for SIMD-style parallelism and some support for task parallelism (the blocks are mostly independent). The missing ingredient for making GPUs truly general purpose accelerators is pipeline-style parallelism. Perhaps a local microcode approach within the blocks could be effective. The central block controller broadcasts instructions, but (subsets of) processors reinterpret those instructions based on their local microcode.

It is also possible that it would be easier to design a high performance, parallelism-style-flexible accelerator starting from something other than a GPU. The reason for the focus on GPUs here is simply the commercial momentum behind them. Millions of developers already have GPUs in their systems to experiment with, which is an important consideration.

8.3.2 Model-based profiling tools

As mentioned in the previous section, one of the biggest challenges we faced in getting new programmers to write good Macah code is that it mostly looks sequential. The results of loop unrolling and pipelining are hard for many programmers to imagine, which makes it hard to gain intuition for why specific dependencies in a program are problematic. We have also run into the kinds of deadlocks described in Chapter 7 in more complex programs.

These problems can be addressed with correctness and performance analysis tools that are designed for accelerator-specific problems. These tools would analyze I/O behavior, workspace memory usage, performance-limiting dependencies, and potentially problematic feedback through streams. The critical requirement for these tools is that they provide feedback in programming language and model terms, not architecture terms.

8.3.3 What's the best auto-tuner?

In Chapter 6 the evaluation of our tuning knob search algorithm focused on the issue of hard-to-predict failures. We have not directly compared our algorithm to other search methods for optimizing a single function. However, given the high cost of testing a configuration and the relatively small number of iterations that are tolerable in many development situations, the question of which search method is fastest in general is an interesting one. The method we developed, based on locally weighted averaging and derivative projection, is somewhat unusual, but we believe it deserves further investigation as a generally efficient auto-tuning search method.

One of the common features of many of the popular approaches to auto-tuning, including direct search-style algorithms, simulated annealing, and genetic algorithms, is that as the search runs there is a notion of *the* current configuration (or a few current configurations in the case of genetic algorithms). The search proceeds by moving in some direction from the current configuration. One of the major problems that all such search algorithms have to contend with is getting trapped in local minima. If all configurations in range of the current configuration are worse than it, the search will stop.

This single configuration style of searching makes a lot of sense for applications like

circuit layout where a single configuration is large, the whole space of possible configurations is gargantuan, and huge portions of the space are very low quality. When a reasonably good configuration is found there are good reasons to not stray too far from it. However, many tuning applications do not have these kinds of dimensions. Most auto-tuning applications have at most a couple tens of knobs; many only two or three. Given this much smaller scale, it is easy to generate a large number of candidate configurations. Also, the fewer dimensions, the easier it is to use trends in a set of test data to predict the value of untested configurations. Clearly more experimentation is needed, but we believe that there is a potential for methods that consider candidate configurations from the whole range of the tuning space for every test.

8.4 The last word

We now have the technology to support C-level programming of a particular family of accelerators. Modest tool and algorithm improvements will soon improve the convenience and speed of accelerator development to the point where most programmers of performance-critical applications should consider using accelerators. Porting between different accelerator families still requires at least a change in coding style for many applications. In practice different languages will be used to program different kinds of accelerators for the near future. The biggest question for the accelerator ecosystem is whether there will continue to be room for multiple families of accelerators with different parallelism-control strengths. If the community converges on a family of architectures that have support for the three fundamental styles of accelerator parallelism control, there will be no need to develop technologies to facilitate porting across the families of accelerators we have today.

BIBLIOGRAPHY

[ACW+09] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. Petabricks: a language and compiler for algorithmic choice. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM.

[ADK+04] Jung Ho Ahn, William J. Dally, Brucek Khailany, Ujval J. Kapasi, and Abhishek Das. Evaluating the Imagine Stream Architecture. In *Proceedings of the 31st annual international symposium on Computer architecture*, page 14, Mnchen, Germany, 2004. IEEE Computer Society.

[AG95] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. Research Report 95/7, Western Research Laboratory, 250 University Avenue Palo Alto, California 94301 USA, September 1995.

[AJ88] R. Allen and S. Johnson. Compiling C for vectorization, parallelization, and inline expansion. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 241–249, New York, NY, USA, 1988. ACM Press.

[And98] Ray Andraka. A survey of CORDIC algorithms for FPGA based computers. In *FPGA '98: Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, pages 191–200, New York, NY, USA, 1998. ACM.

[ATA05] Sitij Agrawal, William Thies, and Saman Amarasinghe. Optimizing stream programs using linear state space analysis. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 126–136, New York, NY, USA, 2005. ACM Press.

[AW77] E. A. Ashcroft and W. W. Wadge. Lucid, a nonprocedural language with iteration. *Commun. ACM*, 20(7):519–526, 1977.

[AWC+10] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. Technical Report MIT-CSAIL-TR-2010-032, Computer Science and ArtificialIntelligence Laboratory, MIT, July 2010.

[BA08] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory models. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 68–78, New York, NY, USA, 2008. ACM.

[BACD97] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 340–347, New York, NY, USA, 1997. ACM.

[Bal94] Shummet Baluja. Population-Based Incremental Learning: A Method for Integrating Genetic Search Based Function Optimization and Competitive Learning. Technical Report CS-94-163, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

[BBKG07] Frank Bouwens, Mladen Berekovic, Andreas Kanstein, and Georgi Gaydadjiev. Architectural Exploration of the ADRES Coarse-Grained Reconfigurable Array. In Springer Berlin / Heidelberg, editor, *Reconfigurable Computing: Architectures, Tools and Applications*, volume Volume 4419/2007 of *Lecture Notes in Computer Science*, pages 1–13, March 2007.

[BDH+00] Reynold Bailey, Delvin Defoe, Ranette Halverson, Richard Simpson, and Nelson Passos. A study of software pipelining for multi-dimensional problems. In *13th International Conference on Parallel and Distributed Computing Systems*, pages 426–431, Las Vegas NV, August 2000.

[BG02] Mihai Budiu and Seth Copen Goldstein. Compiling application-specific hardware. In *International Conference on Field Programmable Logic and Applications (FPL)*, pages 853–863, Montpellier (La Grande-Motte), France, September 2–4 2002.

[BGS94] David Bacon, Susan Graham, and Oliver Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[BGT07] Zachary K. Baker, Maya B. Gokhale, and Justin L. Tripp. Matched Filter Computation on FPGA, Cell and GPU. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 207–218, Washington, DC, USA, 2007. IEEE Computer Society.

[BH93] Thomas Ball and Susan Horwitz. Slicing Programs with Arbitrary Control-flow. In *AADEBUG '93: Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, pages 206–222, London, UK, 1993. Springer-Verlag.

[BHD+02] W. Böhm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar. Mapping a Single Assignment Programming Language to Reconfigurable Systems. *The Journal of Supercomputing*, 21(2):117–130, 2002.

[BJW07] Michael Butts, Anthony Mark Jones, and Paul Wasson. A Structural Object Programming Model, Architecture, Chip and Tools for Reconfigurable Computing. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 55–64, Washington, DC, USA, 2007. IEEE Computer Society.

[BNW+10] John Bodily, Brent Nelson, Zhaoyi Wei, Dah-Jye Lee, and Jeff Chase. A comparison study on implementing optical flow and digital communications on FPGAs and GPUs. *ACM Trans. Reconfigurable Technol. Syst.*, 3(2):1–22, 2010.

[BP09] Gérard Boudol and Gustavo Petri. Relaxed memory models: an operational approach. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 392–403, New York, NY, USA, 2009. ACM.

[BRS07] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. Automatic mapping of nested loops to FPGAs. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 101–111, New York, NY, USA, 2007. ACM.

[BSWG00] Mihai Budiu, Majd Sakr, Kip Walker, and Seth Copen Goldstein. BitValue Inference: Detecting and Exploiting Narrow Bitwidth Computations. In *Europar Conference*. Springer Verlag, 2000.

[Bud03] Mihai Budiu. *Spatial Computation*. PhD thesis, Carnegie Mellon University, Computer Science Department, December 2003. Technical report CMU-CS-03-217.

[Car05] Joäo M. P. Cardoso. Dynamic loop pipelining in data-driven architectures. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 106–115, New York, NY, USA, 2005. ACM.

[CBl] http://en.wikipedia.org/wiki/Blocks_(C_language_extension).

[CBM+93] William Y. Chen, Roger A. Bringmann, Scott A. Mahlke, Sadun Anik, Tokuzo Kiyohara, Nancy J. Warter, Daniel M. Lavery, Wen mei W. Hwu, Richard E. Hank, and John C. Gyllenhaal. Using Profile Information to Assist Advaced Compiler Optimization and Scheduling. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 31–48, London, UK, 1993. Springer-Verlag.

[CCF03] Weihaw Chuang, Brad Calder, and Jeanne Ferrante. Phi-predication for light-weight if-conversion. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 179–190, Washington, DC, USA, 2003. IEEE Computer Society.

[CCH+00] Eylon Caspi, Michael Chu, Randy Huang, Joseph Yeh, John Wawrzynek, and André DeHon. Stream Computations Organized for Reconfigurable Execution (SCORE). In *International Conference on Field-Programmable Logic and Applications*, pages 605–614, London, UK, August 2000. Springer-Verlag.

[CCH05] Chun Chen, Jacqueline Chame, and Mary Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 111–122, Washington, DC, USA, 2005. IEEE Computer Society.

[CCL+98] Bradford L. Chamberlain, Sung-Eun Choi, E Christopher Lewis, Lawrence Snyder, W. Derrick Weathersby, and Calvin Lin. The Case for High-Level Parallel Programming in ZPL. *IEEE Comput. Sci. Eng.*, 5(3):76–86, 1998.

[CCZ07] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[CDG+06] Albert Cohen, Sébastien Donadio, Maria-Jesus Garzaran, Christoph Herrmann, Oleg Kiselyov, and David Padua. In search of a program generator to implement generic transformations for high-performance computing. *Science of Computer Programming*, 62(1):25–46, September 2006.

[CE06] Allan Carroll and Carl Ebeling. Reducing the Space Complexity of Pipelined Routing Using Modified Range Encoding. In *Proceedings of the International Conference on Field Programmable Logic and Applications*, September 2006.

[Cel04] Celoxica. *Handel-C Language Reference Manual RM-1003-4.2*. Celoxica, 2004.

[CFBE98] Darren C. Cronquist, Paul Franklin, Stefan G. Berg, and Carl Ebeling. Specifying and Compiling Applications for RaPiD. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 116–125. IEEE Computer Society Press, 1998.

[CFF+99] D.C. Cronquist, C. Fisher, M. Figueroa, P. Franklin, and C. Ebeling. Architecture design of reconfigurable pipelined datapaths. In *Advanced Research in VLSI, 1999. Proceedings. 20th Anniversary Conference on*, pages 23–40, Atlanta, 1999.

[CFS90] Ron Cytron, Jeanne Ferrante, and V. Sarkar. Compact representations for control dependence. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 337–351, New York, NY, USA, 1990. ACM.

[CGH+05] Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. ACME: adaptive compilation made efficient. *SIGPLAN Not.*, 40(7):69–77, 2005.

[CH99] Mark L. Chang and Scott Hauck. Adaptive Computing in NASA Multi-Spectral Image Processing. In *Military and Aerospace Applications of Programmable Devices and Technologies International Conference*, 1999.

[CH04] I-Hsin Chung and Jeffrey K. Hollingsworth. Using information from prior runs to improve automated tuning systems. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 30, Washington, DC, USA, 2004. IEEE Computer Society.

[CHW00] T. Callahan, J. Hauser, and J. Wawrzynek. The Garp Architecture and C Compiler. *IEEE Computer*, 33(4):62–69, 2000.

[CLS+08] Shuai Che, Jie Li, Jeremy W. Sheaffer, Kevin Skadron, and John Lach. Accelerating compute-intensive applications with GPUs and FPGAs. In *SASP '08: Proceedings of the 2008 Symposium on Application Specific Processors*, pages 101–107, Washington, DC, USA, 2008. IEEE Computer Society.

[CMmWH91] Pohua P. Chang, Scott A. Mahlke, and Wen mei W. Hwu. Using Profile Information to Assist Classic Code Optimizations. *Softw. Pract. Exper.*, 21(12):1301–1321, 1991.

[Cra] Inc. Cray. Cray XD1 Supercomputer Overview. http://www.cray.com/products/xd1/.

[CSJC10] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. A dynamically configurable coprocessor for convolutional neural networks. In *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*, pages 247–257, New York, NY, USA, 2010. ACM.

[CSS99] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *LCTES '99: Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, pages 1–9, New York, NY, USA, 1999. ACM.

[CST02] Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *J. Supercomput.*, 23(1):7–22, 2002.

[CW00] Timothy J. Callahan and John Wawrzynek. Adapting Software Pipelining for Reconfigurable Computing. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES) 2000*, 2000.

[DCF+07] Christophe Dubach, John Cavazos, Björn Franke, Grigori Fursin, Michael F.P. O'Boyle, and Olivier Temam. Fast compiler optimisation evaluation using code-feature based performance prediction. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 131–142, New York, NY, USA, 2007. ACM Press.

[DLD+03] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and Ujval J. Kapasi. Merrimac: Supercomputing with Streams. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 35, Washington, DC, USA, 2003. IEEE Computer Society.

[FCOT05] Grigori Fursin, Albert Cohen, M. O'Boyle, and Olivier Temam. A Practical Method For Quickly Evaluating Program Optimizations. In *Intl. Conf. on High Performance Embedded Architectures and Compilers (HiPEAC'05)*, number 3793 in LNCS, pages 29–46, Barcelona, Spain, November 2005. Springer-Verlag.

[FCT07] Mohammed Fellahi, Albert Cohen, and Sid Touati. Code-size conscious pipelining of imperfectly nested loops. In *MEDEA '07: Proceedings of the 2007 workshop on MEmory performance*, pages 49–55, New York, NY, USA, 2007. ACM.

[FCVE+09] Stephen Friedman, Allan Carroll, Brian Van Essen, Benjamin Ylvisaker, Carl Ebeling, and Scott Hauck. SPR: an architecture-adaptive CGRA mapping tool. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 191–200, New York, NY, USA, 2009. ACM.

[FDF98] Paolo Faraboschi, Giuseppe Desoli, and Joseph A. Fisher. Clustered Instruction-Level Parallel Processors. Technical Report HPL-98-204, Hewlett Packard Laboratories Cambridge, December 1998.

[FH05] T.W. Fry and S.A. Hauck. SPIHT image compression on FPGAs. *IEEE Transactions on Circuits and Systems for Video Technology*, 15(9):1138–1147, 2005.

[Fis81] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput.*, 30(7):478–490, 1981.

[FJ05] Matteo Frigo and Steven G. Johnson. The Design and Implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[FLC03] M Forina, S Lanteri, and C Casolino. Cluster analysis: significance, empty space, clustering tendency, non-uniformity. II - empty space index. *Annali di Chimica*, 93(5-6):489–498, May-June 2003.

[FOK02] Grigori Fursin, Michael F. P. O'Boyle, and Peter M. W. Knijnenburg. Evaluating Iterative Compilation. In William Pugh and Chau-Wen Tseng, editors, *LCPC*, volume 2481 of *Lecture Notes in Computer Science*, pages 362–376. Springer, 2002.

[FOTF05] Björn Franke, Michael O'Boyle, John Thomson, and Grigori Fursin. Probabilistic source-level optimisation of embedded programs. *SIGPLAN Not.*, 40(7):78–86, 2005.

[Gan94] Amit Ganesh. Fusing loops with backward inter loop data dependence. *SIGPLAN Not.*, 29(12):25–30, 1994.

[GBL10] Cristian Grozea, Zorana Bankovic, and Pavel Laskov. FPGA vs. multi-core CPUs vs. GPUs: Hands-on experience with a sorting application. In *Facing the Multi-Core Challenge: Conference for Young Scientists at the Heidelberger Akademie der Wissenschaften*, 2010.

[gcc] http://gcc.gnu.org/onlinedocs/gcc/Nested-Functions.html.

[GCHP02] Benjamin Goldberg, Emily Crutcher, Chad Huneycutt, and Krishna V. Palem. Software bubbles: Using predication to compensate for aliasing in software pipelines. In *PACT '02: Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 211–221, Washington, DC, USA, 2002. IEEE Computer Society.

[GDKG05] M.D. Galanis, G. Dimitroulakos, A.P. Kakarountas, and C.E. Goutis. Speedups from partitioning software kernels to fpga hardware in embedded socs. In *Signal Processing Systems Design and Implementation, 2005. IEEE Workshop on*, pages 485 – 490, 2005.

[GF95] Anwar M. Ghuloum and Allan L. Fisher. Flattening and parallelizing irregular, recurrent loop nests. *SIGPLAN Not.*, 30(8):58–67, 1995.

[GFM+03] Maya Gokhale, Janette Frigo, Kevin McCabe, James Theiler, Christophe Wolinski, and Dominique Lavenier. Experience with a Hybrid Processor: K-Means Clustering. *The Journal of Supercomputing*, 26(2):131–148, 2003.

[GGHvdG01] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. Flame: Formal linear algebra methods environment. *ACM Trans. Math. Softw.*, 27(4):422–455, 2001.

[GMR99] Phillip B. Gibbons, Yossi Matias, and Vijaya Ramachandran. The queue-read queue-write pram model: Accounting for contention in parallel algorithms. *SIAM J. Comput.*, 28:733–769, February 1999.

[GRE+01] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[GS98] M. Gokhale and J. Stone. NAPA C: Compiling for Hybrid RISC/FPGA Architecture. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1998.

[GSAK00] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski. Stream-oriented FPGA computing in the Streams-C high level language. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 49–56, 2000.

[GSB+00] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matt Moe, and Reed Taylor. PipeRench: A Reconfigurable Architecture and Compiler. *IEEE Computer*, 33(4):70–77, 2000.

[GSM+99] Seth Copen Goldstein, Herman Schmit, Matthew Moe, Mihai Budiu, Srihari Cadambi, R. Reed Taylor, and Ronald Laufer. PipeRench: A Coprocessor for Streaming Multimedia Acceleration. In *26th International Symposium on Computer Architecture (ISCA99)*, 1999.

[GSZ01] Elana Granston, Eric Stotzer, and Joe Zbiciak. Software pipelining irregular loops on the tms320c6000 vliw dsp architecture. In *LCTES '01: Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*, pages 138–144, New York, NY, USA, 2001. ACM.

[GTK+02] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. A stream compiler for communication-exposed architectures. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 291–303, New York, NY, USA, 2002. ACM.

[GYQ10] Jichi Guo, Qing Yi, and Apan Qasem. Evaluating the role of optimization-specific search heuristics in effective autotuning. Technical Report CS-TR-2010-010, University of Texas at San Antonio, July 2010.

[GZD+00] Daniel D. Gajski, Jianwen Zhu, Rainer Domer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Methodology*. Springer, March 2000.

[HJL+07] B. Harris, A.C. Jacob, J.M. Lancaster, J. Buhler, and R.D. Chamberlain. A banded smith-waterman FPGA accelerator for mercury BLASTP. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 765 –769, aug. 2007.

[HML+09] Michael Haselman, Robert Miyaoka, Thomas K. Lewellen, Scott Hauck, Wendy McDougald, and Don Dewitt. FPGA-based front-end electronics for positron emission tomography. In *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 93–102, New York, NY, USA, 2009. ACM.

[HNC+01] Malay Haldar, Anshuman Nayak, Alok Choudhary, Prith Banerjee, and Nagraj Shenoy. Fpga hardware synthesis from matlab. In *VLSID '01: Proceedings of the The 14th International Conference on VLSI Design (VLSID '01)*, page 299, Washington, DC, USA, 2001. IEEE Computer Society.

[HNS09] Albert Hartono, Boyana Norris, and P. Sadayappan. Annotation-based empirical performance tuning using orio. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–11, Washington, DC, USA, 2009. IEEE Computer Society.

[HR92] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *ICSE '92: Proceedings of the 14th international conference on Software engineering*, pages 392–411, New York, NY, USA, 1992. ACM.

[HU05] K. Scott Hemmert and Keith D. Underwood. An Analysis of the Double-Precision Floating-Point FFT on FPGAs. In *FCCM '05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pages 171–180, Washington, DC, USA, 2005. IEEE Computer Society.

[HW97] John R. Hauser and John Wawrzynek. A MIPS Processor with a Reconfigurable Coprocessor. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 12–21. IEEE Computer Society Press, 1997. Los Alamitos, CA.

[JTLC09] Qiwei Jin, David B. Thomas, Wayne Luk, and Benjamin Cope. Exploring reconfigurable architectures for tree-based option pricing models. *ACM Trans. Reconfigurable Technol. Syst.*, 2(4):1–17, 2009.

[JXHX02] A.K. Jain, Xiaowei Xu, Tin Kam Ho, and Fan Xiao. Uniformity testing using minimal spanning tree. In *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, volume 4, pages 281–284 vol.4, 2002.

[KDK+01] Brucek Khailany, William J. Dally, Ujval J. Kapasi, Peter Mattson, Jinyung Namkoong, John D. Owens, Brian Towles, Andrew Chang, and Scott Rixner. Imagine: Media Processing with Streams. *IEEE Micro*, 21(2):35–46, 2001.

[KFM06] Manjunath Kudlur, Kevin Fan, and Scott Mahlke. Streamroller: Automatic Synthesis of Prescribed Throughput Accelerator Pipelines. In *International Conference on Hardware/Software Codesign and System Synthesis*, October 2006.

[KHH+04] Prasad Kulkarni, Stephen Hines, Jason Hiser, David Whalley, Jack Davidson, and Douglas Jones. Fast searches for effective optimization phase sequences. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 171–182, New York, NY, USA, 2004. ACM.

[KKLW80] D. J. Kuck, R. H. Kuhn, B. Leasure, and M. Wolfe. The structure of an advanced vectorizer for pipelined processors. In *Proc. IEEE Computer Society Fourth International Computer Software and Applications Conf.*, October 1980.

[KLT03] Tamara G. Kolda, Robert Michael Lewis, and Virginia Torczon. Optimization by direct search: New perspectives on some classical and modern methods. *SIAM Review*, 45:385–482, 2003.

[KM94] Ken Kennedy and Kathryn S. McKinley. Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 301–320, London, UK, 1994. Springer-Verlag.

[Kni98] Peter M.W. Knijnenburg. Flattening VLIW code generation for imperfectly nested loops. Technical report, Department of Computer Science, Leiden University, 1998.

[KNP08] Arun Kejariwal, Alexandru Nicolau, and Constantine D. Polychronopoulos. Enhanced loop coalescing: A compiler technique for transforming non-uniform iteration spaces. *Lecture Notes in Computer Science*, Volume 4759/2009:17–32, January 2008.

[KRD+03] Ujval J. Kapasi, Scott Rixner, William J. Dally, Brucek Khailany, Jung Ho Ahn, Peter Mattson, and John D. Owens. Programmable Stream Processors. *IEEE Computer*, 36(8):54–62, 2003.

[KSP09] Thomas Karcher, Christoph Schaefer, and Victor Pankratius. Auto-tuning support for manycore applications: perspectives for operating systems and compilers. *SIGOPS Oper. Syst. Rev.*, 43(2):96–97, 2009.

[KZM+03] Prasad Kulkarni, Wankang Zhao, Hwashin Moon, Kyunghwan Cho, David Whalley, Jack Davidson, Mark Bailey, Yunheung Paek, and Kyle Gallivan. Finding effective optimization phase sequences. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 12–23, New York, NY, USA, 2003. ACM.

[Lam88] Monica Lam. Software pipelining: an effective scheduling technique for VLIW machines. In *ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 318–328, New York, NY, USA, 1988. ACM Press.

[LCD91] David Levine, David Callahan, and Jack Dongarra. A comparative study of automatic vectorizing compilers. *Parallel Computing*, 17(10-11):1223–1244, 1991.

[LE04] Song Li and C. Ebeling. QuickRoute: a fast routing algorithm for pipelined architectures. In *IEEE International Conference on Field-Programmable Technology*, pages 73–80, Queensland, Australia, 2004.

[LGAV96] Josep Llosa, Antonio González, Eduard Ayguadé, and Mateo Valero. Swing modulo scheduling: A lifetime-sensitive approach. In *PACT '96: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*, page 80, Washington, DC, USA, 1996. IEEE Computer Society.

[LKSK08] Changhee Lee, Donguk Kim, Hayong Shin, and Deok-Soo Kim. Trash removal algorithm for fast construction of the elliptic gabriel graph using delaunay triangulation. *Comput. Aided Des.*, 40(8):852–862, 2008.

[LLS98] E. Christopher Lewis, Calvin Lin, and Lawrence Snyder. The implementation and evaluation of fusion and contraction in array languages. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 50–59, New York, NY, USA, 1998. ACM.

[LS03] Benjamin A. Levine and Herman H. Schmit. Efficient Application Representation for HASTE: Hybrid Architectures with a Single, Transformable Executable. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, page 101, Washington, DC, USA, 2003. IEEE Computer Society.

[LW10] Anders Logg and Garth N. Wells. Dolfin: Automated finite element computing. *ACM Trans. Math. Softw.*, 37(2):1–28, 2010.

[LZSS04] Meilin Liu, Qingfeng Zhuge, Zili Shao, and Edwin H.-M. Sha. General loop fusion technique for nested loops considering timing and code size. In *CASES '04: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 190–201, New York, NY, USA, 2004. ACM.

[Mat01] Peter Mattson. *A Programming System for the Imagine Media Processor*. PhD thesis, Stanford University, 2001.

[MD01] Kalyan Muthukumar and Gautam Doshi. Software pipelining of nested loops. In *CC '01: Proceedings of the 10th International Conference on Compiler Construction*, pages 165–181, London, UK, 2001. Springer-Verlag.

[ME95] Larry McMurchie and Carl Ebeling. PathFinder: A negotiation-based performance-driven router for FPGAs. In *ACM International Symposium on Field-Programmable Gate Arrays*, pages 111–117. ACM Press, 1995. Monterey, California, United States.

[MGAK03] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[MHM+95] Scott A. Mahlke, Richard E. Hank, James E. McCormick, David I. August, and Wen-Mei W. Hwu. A comparison of full and partial predicated execution support for ilp processors. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 138–150, New York, NY, USA, 1995. ACM.

[MJ02] Dragan Milicev and Zoran Jovanovic. Control flow regeneration for software pipelined loops with conditions. *Int. J. Parallel Program.*, 30(3):149–179, 2002.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *POPL*, 2005.

[MSBL98] Andrew Moore, Jeff Schneider, Justin Boyan, and Mary Soon Lee. Q2: Memory-based active learning for optimizing noisy continuous functions. In J. Shavlik, editor, *Proceedings of the Fifteenth International Conference of Machine Learning*, pages 386–394, San Francisco, CA, 1998. Morgan Kaufmann.

[MSBSV93] Patrick McGeer, Jagesh Sanghavi, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. ESPRESSO-signature: A new exact minimizer for logic functions. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 1(3):432–440, December 1993.

[MUS05] Krishna Muriki, Keith D. Underwood, and Ron Sass. RC-BLAST: Towards a Portable, Cost-Effective Open Source Hardware Implementation. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 7*, page 196.2, Washington, DC, USA, 2005. IEEE Computer Society.

[MVV+02] Bingfeng Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. DRESC: a retargetable compiler for coarse-grained reconfigurable architectures. In *IEEE International Conference on Field-Programmable Technology*, pages 166–173, 2002.

[MVV+03] Bingfen Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix. In *International Conference on Field-Programmable Logic and Applications*, volume 2778, pages 61–70, Lisbon, Portugal, 2003. 2003.

[NBH+08] Y.L. Nelson, B. Bansal, M. Hall, A. Nakano, and K. Lerman. Model-guided performance tuning of parameter values: A case study with molecular dynamics visualization. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008.

[NM65] J. A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[NMRW02] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

[OD93] M. T. O'Keefe and H. G. Dietz. Loop Coalescing and Scheduling for Barrier MIMD Architectures. *IEEE Trans. Parallel Distrib. Syst.*, 4(9):1060–1064, September 1993.

[Pan01] Preeti Ranjan Panda. SystemC: a modeling platform supporting multiple design abstractions. In *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, pages 75–80, New York, NY, USA, 2001. ACM.

[PBD+08] A. Putnam, D. Bennett, E. Dellinger, J. Mason, P. Sundararajan, and S. Eggers. Chimps: A c-level compilation flow for hybrid cpu-fpga architectures. In *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pages 173 –178, 2008.

[PCB02] C. J. Price, I. D. Coope, and D. Byatt. A convergent variant of the nelder-mead algorithm. *J. Optim. Theory Appl.*, 113(1):5–19, 2002.

[PD10] Jongsoo Park and William J. Dally. Buffer-space efficient and deadlock-free scheduling of stream applications on multi-core architectures. In *SPAA '10: Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 1–10, New York, NY, USA, 2010. ACM.

[PE06] Zhelong Pan and Rudolf Eigenmann. Fast, automatic, procedure-level performance tuning. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 173–181, New York, NY, USA, 2006. ACM Press.

[PGS+09a] Alexandros Papakonstantinou, Karthik Gururaj, John A. Stratton, Deming Chen, Jason Cong, and Wen-Mei W. Hwu. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. *Application Specific Processors, Symposium on*, 0:35–42, 2009.

[PGS+09b] Alexandros Papakonstantinou, Karthik Gururaj, John A. Stratton, Deming Chen, Jason Cong, and Wen-Mei W. Hwu. High-performance cuda kernel execution on fpgas. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 515–516, New York, NY, USA, 2009. ACM.

[PHA02] Darin Petkov, Randolph E. Harr, and Saman P. Amarasinghe. Efficient pipelining of nested loops: Unroll-and-squash. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 136, Washington, DC, USA, 2002. IEEE Computer Society.

[PKCD05] Karl Papadantonakis, Nachiket Kapre, Stephanie Chan, and Andr DeHon. Pipelining Saturated Accumulation. In *IEEE International Conference on Field-Programmable Technology*, pages 19–26. IEEE, December 2005.

[PMJ+05] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code Generation for DSP Transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

[Pol87] Constantine D. Polychronopoulos. Loop Coalescing: A Compiler Transformation for Parallel Machines. In *Proc. International Conf. on Parallel Processing*, pages 235–242, August 1987.

[Poz05] D.S. Poznanovic. Application development on the src computers, inc. systems. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 78a – 78a, 2005.

[PSC06] Joon C. Park, Hayong Shin, and Byoung K. Choi. Elliptic gabriel graph for finding neighbors in a point set and its application to normal vector estimation. *Comput. Aided Des.*, 38(6):619–626, 2006.

[PT05] David Pellerin and Scott Thibault. *Practical FPGA Programming in C*. Prentice Hall PTR, April 2005.

[PW86] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Commun. ACM*, 29(12):1184–1201, 1986.

[QCS02] Yi Qian, Steve Carr, and Philip Sweany. Loop fusion for clustered VLIW architectures. In *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 112–119, New York, NY, USA, 2002. ACM.

[QKMC06] Apan Qasem, Ken Kennedy, and John Mellor-Crummey. Automatic tuning of whole applications using direct search and a performance-based transformation system. *J. Supercomput.*, 36(2):183–196, 2006.

[Ram94] J. Ramanujam. Optimal software pipelining of nested loops. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 335–342, Washington, DC, USA, 1994. IEEE Computer Society.

[Rau94a] B. Rau. Iterative Modulo Scheduling. Technical Report Technical Report HPL-94-115, HP Labs, 1994.

[Rau94b] B. Ramakrishna Rau. Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops. In *International Symposium on Microarchitecture*, pages 63–74. ACM Press, 1994. San Jose, California, United States.

[RPH+08] Manman Ren, Ji Young Park, Mike Houston, Alex Aiken, and William J. Dally. A tuning framework for software-managed memory hierarchies. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 280–291, New York, NY, USA, 2008. ACM.

[RS01] S. Ramachandran and S. Srinivasan. FPGA implementation of a novel, fast motion estimation algorithm for real-time video compression. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 213–219, New York, NY, USA, 2001. ACM Press.

[RTG+04] Hongbo Rong, Zhizhong Tang, R. Govindarajan, Alban Douillet, and Guang R. Gao. Single-dimension software pipelining for multi-dimensional loops. In *CGO '04: Proceedings of the international symposium on Code*

generation and optimization, page 163, Washington, DC, USA, 2004. IEEE Computer Society.

[RTG+07] Hongbo Rong, Zhizhong Tang, R. Govindarajan, Alban Douillet, and Guang R. Gao. Single-dimension software pipelining for multidimensional loops. *ACM Trans. Archit. Code Optim.*, 4(1):7, 2007.

[RVDB10] Sean Rul, Hans Vandierendonck, Joris D'Haene, and Koen De Bosschere. An experimental study on performance portability of opencl kernels. In *Application Accelerators in High Performance Computing*, 2010.

[RW06] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. Massachusetts Institute of Technology Press, 2006.

[SAMO03] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 77–90, New York, NY, USA, 2003. ACM Press.

[SBA00] Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bidwidth analysis with application to silicon compilation. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 108–120, New York, NY, USA, 2000. ACM.

[Sch09] Christoph A. Schaefer. Reducing search space of auto-tuners using parallel patterns. In *IWMSE '09: Proceedings of the 2009 ICSE Workshop on Multicore Software Engineering*, pages 17–24, Washington, DC, USA, 2009. IEEE Computer Society.

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[SGL96] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Identifying loops using DJ graphs. *ACM Trans. Program. Lang. Syst.*, 18(6):649–658, 1996.

[SMDL03] Mikhail Smelyanskiy, Scott A. Mahlke, Edward S. Davidson, and Hsien-Hsin S. Lee. Predicate-aware scheduling: a technique for reducing resource constraints. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 169–178, Washington, DC, USA, 2003. IEEE Computer Society.

[Smi91] Lauren L. Smith. Vectorizing C compilers: how good are they? In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 544–553, New York, NY, USA, 1991. ACM Press.

[Sny86] Lawrence Snyder. *Type architectures, shared memory, and the corollary of modest potential*. Annual Reviews Inc., Palo Alto, CA, USA, 1986.

[SP06] Ronald Scrofano and Viktor K. Prasanna. Preliminary Invesitgation of Advanced Electrostatics in Molecular Dynamics on Reconfigurable Computers. In *Supercomputing, 2006. Proceedings of the ACM/IEEE SC 2006 Conference*, November 2006.

[SPT09] Christoph A. Schaefer, Victor Pankratius, and Walter F. Tichy. Atune-il: An instrumentation language for auto-tuning parallel applications. In Springer Berlin / Heidelberg, editor, *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, volume LNCS, pages 9–20, August 2009.

[SS01] Bernhard Schlkopf and Alexander J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. The MIT Press, 2001.

[Ste06] Mark W. Stephenson. *Automating the Construction of Compiler Heuristics Using Machine Learning*. PhD thesis, Massachusetts Institute of Technology, June 2006.

[Sul03] David Gerard Sullivan. *Using Probabilistic Reasoning to Automate Software Tuning*. PhD thesis, Harvard University, Cambridge, Massachusetts, September 2003.

[TCC+09] A. Tiwari, Chun Chen, J. Chame, M. Hall, and J.K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, May 2009.

[TCK09] Dimitris Theodoropoulos, Catalin Bogdan Ciobanu, and Georgi Kuzmanov. Wave field synthesis for 3d audio: architectural prospectives. In *CF '09: Proceedings of the 6th ACM conference on Computing frontiers*, pages 127–136, New York, NY, USA, 2009. ACM.

[TCMC08] Kieron Turkington, George A. Constantinides, Konstantinos Masselos, and Peter Y. K. Cheung. Outer loop pipelining for application specific datapaths in FPGAs. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(10):1268–1280, Oct. 2008.

[THL09] David Barrie Thomas, Lee Howes, and Wayne Luk. A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In *FPGA '09: Proceeding of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 63–72, New York, NY, USA, 2009. ACM.

[TJH02] Justin L. Tripp, Preston A. Jackson, and Brad L. Hutchings. *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, volume 2438/2002 of *Lecture Notes in Computer Science*, chapter Sea Cucumber: A Synthesizing Compiler for FPGAs, pages 875–885. Springer Berlin / Heidelberg, January 2002.

[TKA02] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Computational Complexity*, pages 179–196, 2002.

[TKM+02] M.B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, Jae-Wook Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs. *IEEE Micro*, 22(2):25–35, 2002.

[TKS+05] William Thies, Michal Karczmarek, Janis Sermulins, Rodric Rabbah, and Saman Amarasinghe. Teleport messaging for distributed stream programs. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 224–235, New York, NY, USA, 2005. ACM.

[TPA+05] J.L. Tripp, K.D. Peterson, C. Ahrens, J.D. Poznanovic, and M. Gokhale. Trident: An FPGA Compiler Framework for Floating-Point Algorithms. In *International Workshop on Field Programmable Logic and Applications*, 2005.

[TPO05] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: simplified programming of graphics-processing units for general-purpose uses via data-parallelism. Technical Report MSR-TR-2004-184, Microsoft Corporation, December 2005.

[TTH09] Ananta Tiwari, Vahid Tabatabaee, and Jeffrey K. Hollingsworth. Tuning parallel applications in parallel. *Parallel Comput.*, 35(8-9):475–492, 2009.

[TVVA03] Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 204–215, Washington, DC, USA, 2003. IEEE Computer Society.

[VBCG04] Girish Venkataramani, Mihai Budiu, Tiberiu Chelcea, and Seth Goldstein. C to Asynchronous Dataflow Circuits: An End-to-End Toolflow. In *International Workshop on Logic and Synthesis (IWLS)*, June 2004.

[VDY05] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing.

[vHK92] Reinhard von Hanxleden and Ken Kennedy. Relaxing SIMD control flow constraints using loop transformations. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 188–199, New York, NY, USA, 1992. ACM.

[vW97] R. A. van de Geijn and J. Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[VWC+09] B. Van Essen, A. Wood, A. Carroll, S. Friedman, R. Panda, B. Ylvisaker, C. Ebeling, and S. Hauck. Static versus scheduled interconnect in Coarse-Grained Reconfigurable Arrays. In *International Conference on Field-Programmable Logic and Applications*, pages 268–275, 31 2009-Sept. 2 2009.

[Wan04] Albert Wang. The Stretch Architecture: Raising the Level of Productivity and Compute Efficiency. Keynote Speech, 6th WorkShop on Media and Streaming Processors, December 2004.

[Wei81] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[WHSB92] Nancy J. Warter, Grant E. Haab, Krishna Subramanian, and John W. Bockhaus. Enhanced modulo scheduling for loops with conditional branches. In *MICRO 25: Proceedings of the 25th annual international symposium on Microarchitecture*, pages 170–179, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[WMHR93] Nancy J. Warter, Scott A. Mahlke, Wen-Mei W. Hwu, and B. Ramakrishna Rau. Reverse if-conversion. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 290–299, New York, NY, USA, 1993. ACM.

[WMPW03] Nicholas Weaver, Yury Markovskiy, Yatish Patel, and John Wawrzynek. Post-placement C-slow retiming for the Xilinx Virtex FPGAs. In *FPGA '03:*

Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays, pages 185–194, New York, NY, USA, 2003. ACM Press.

[WPD01] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).

[WWK+01] Perry H. Wang, Hong Wang, Ralph M. Kling, Kalpana Ramakrishnan, and John P. Shen. Register renaming and scheduling for dynamic execution of predicated code. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 15, Washington, DC, USA, 2001. IEEE Computer Society.

[YLR+03] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A comparison of empirical and model-driven optimization. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 63–76, New York, NY, USA, 2003. ACM Press.

[YMA+06] Zhiyi Yu, Michael Meeuwsen, Ryan Apperson, Omar Sattari, Michael Lai, Jeremy Webb, Eric Work, Tinoosh Mohsenin, Mandeep Singh, and Bevan M. Baas. An Asynchronous Array of Simple Processors for DSP Applications. In *IEEE International Solid-State Circuits Conference, (ISSCC '06)*, February 2006.

[YPS05] Kamen Yotov, Keshav Pingali, and Paul Stodghill. Think globally, search locally. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 141–150, New York, NY, USA, 2005. ACM Press.

[YSD06] Haihang You, Keith Seymour, and Jack Dongarra. An effective empirical search method for automatic software tuning. Technical Report ICL-UT-05-02, Dept. of Computer Science, University of Tennessee, 2006.

[YSY+07] Qing Yi, Keith Seymour, Haihang You, Richard Vuduc, and Dan Quinlan. POET: Parameterized optimizations for empirical tuning. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS*, pages 1–8, March 2007.

[YTZL97] T. Yu, Z. Tang, C. Zhang, and J. Luo. Control mechanism for software pipelining on nested loop. In *APDC '97: Proceedings of the 1997 Advances*

in Parallel and Distributed Computing Conference (APDC '97), page 345, Washington, DC, USA, 1997. IEEE Computer Society.

[YVE06] Benjamin Ylvisaker, Brian Van Essen, and Carl Ebeling. A Type Architecture for Hybrid Micro-Parallel Computers. In *IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 99–110. IEEE, April 2006.

[YW07] Qing Yi and R. Clint Whaley. Automated transformation for performance-critical kernels. In *LCSD '07: Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 109–119, New York, NY, USA, 2007. ACM.

[ZCS08] Fang Zhong, D.W. Capson, and D.C. Schuurman. Parallel architecture for PCA image feature detection using FPGA. In *Electrical and Computer Engineering, 2008. CCECE 2008. Canadian Conference on*, pages 001341 – 001344, may. 2008.

[ZHCC09] Hans Zima, Mary Hall, Chun Chen, and Jaqueline Chame. Model-guided autotuning of high-productivity languages for petascale computing. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 151–166, New York, NY, USA, 2009. ACM.

[ZK05a] Yuan Zhao and K. Kennedy. Scalarization on short vector machines. In *ISPASS '05: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, pages 187–196, Washington, DC, USA, 2005. IEEE Computer Society.

[ZK05b] Yuan Zhao and Ken Kennedy. Scalarization using loop alignment and loop skewing. *J. Supercomput.*, 31(1):5–46, 2005.

[ZLCP04] Ce Zhu, Xiao Lin, Lappui Chau, and Lai-Man Po. Enhanced Hexagonal Search for Fast Block Motion Estimation. *IEEE Transactions on Circuits and Systems for Video Technology*, 14(10):1210–1214, October 2004.

[ZXQ+08] Qingfeng Zhuge, Chun Jason Xue, Meikang Qiu, Jingtong Hu, and Edwin H.-M. Sha. Timing optimization via nest-loop pipelining considering code size. *Microprocessors and Microsystems*, 32(7):351 – 363, 2008.

[ZZH+09] Dan Zhang, Rongcai Zhao, Lin Han, Tao Wang, and Jin Qu. An implementation of Viterbi algorithm on GPUs. In *ICISE '09: Proceedings of the 2009 First IEEE International Conference on Information Science and Engineering*, pages 121–124, Washington, DC, USA, 2009. IEEE Computer Society.

Appendix A

TUNING DATA

This appendix contains more detailed plots from our tuning knob experiments. The first set of plots shows the performance of the three search strategies (full tuning knob, tuning knob with trivial failures, random) individually for each application/architecture combination. As a reminder, each application was compiled and run on four different simulated architectures that were defined by two parameters: size of the architecture (large or small) and number of embedded memories (many or few). Like the performance plot in Chapter 6, the 10th/90th percentile range is plotted.

The next group of plots show a variety of important program factors for each application/architecture combination. The meanings of the tuning knobs for the applications are described in Section 6.10.1. The meanings of the symbols in these plots are shown directly below.

Symbol	Meaning
Black dot	Not tested
Green star	Max initiation interval failure
Empty blue square	Data memory or I/O failure
Red X	Compiler timeout failure
Filled square	Color indicates normalized value of the feature

In contrast to the performances plot shown in Chapter 6, some of the features shown here have values even for configurations that fail, which means that some coordinates have both a failure symbol and a value color.

The features presented are listed in the table below. All values are normalized so that the smallest value in each plot is 0 and the largest is 1.

Feature	Comments
Performance	Run time of the application
DFG Size	A measure of the intermediate representation size. This is used as the proxy metric for predicting compiler time-outs.
Req'd Arrays	Number of embedded memories required
Req'd In Ports	Number of input ports required
Req'd Out Ports	Number of input ports required
II	Initiation interval. See Chapter 4 for description.

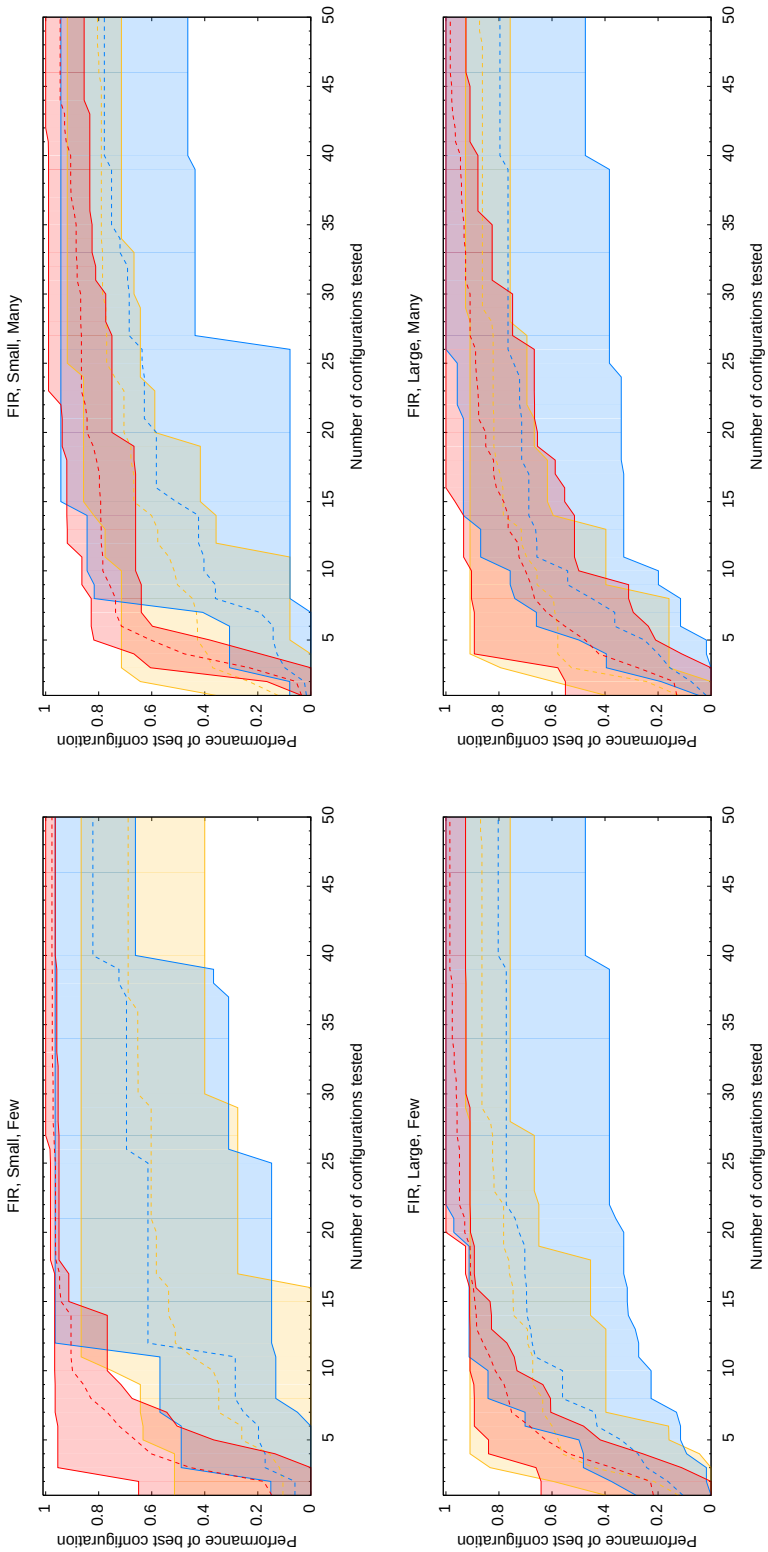


Figure A.1: Quality of best configuration found as a function of number of tests for the FIR filter application.

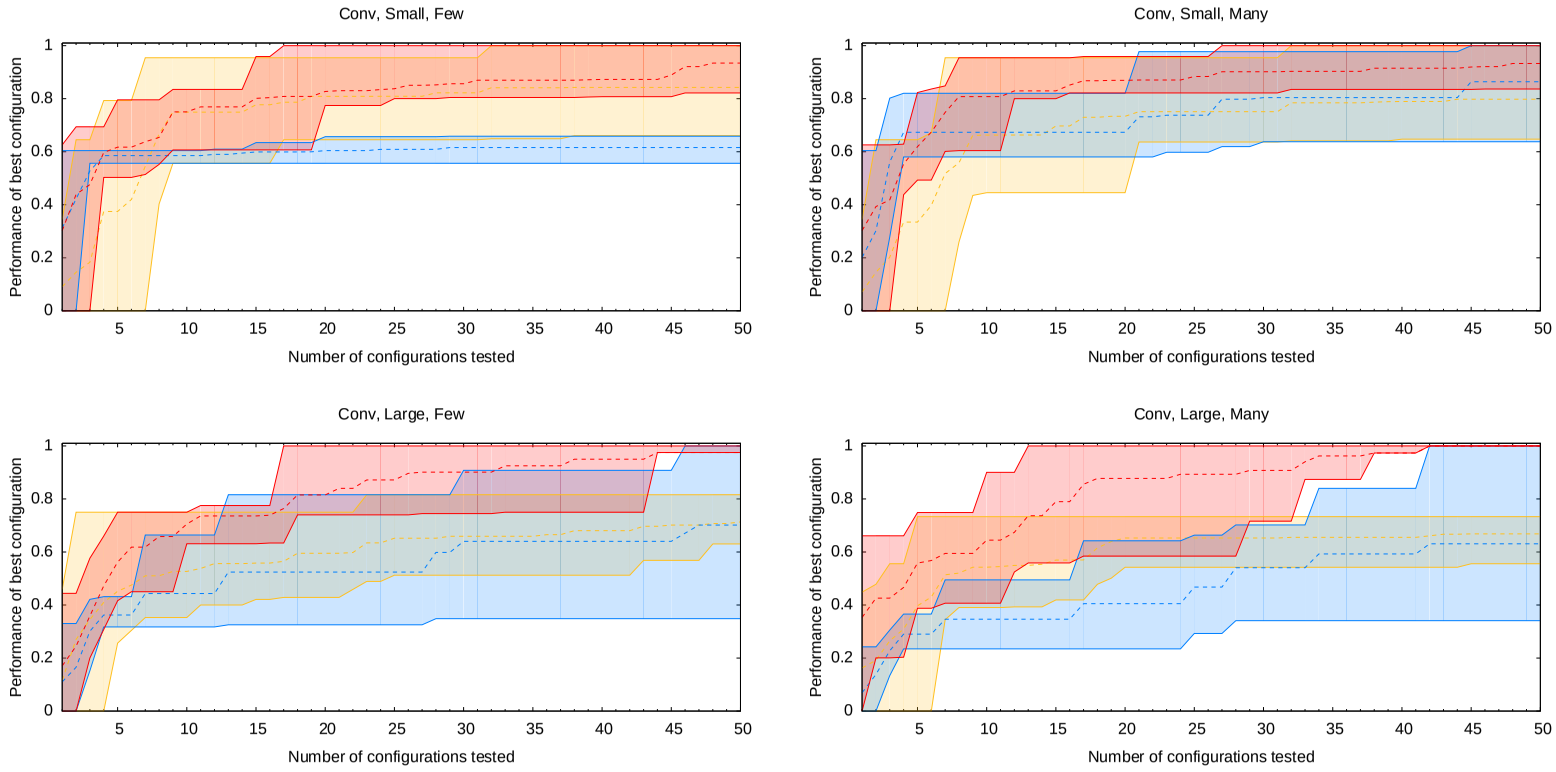


Figure A.2: Quality of best configuration found as a function of number of tests for the 2D convolution filter application.

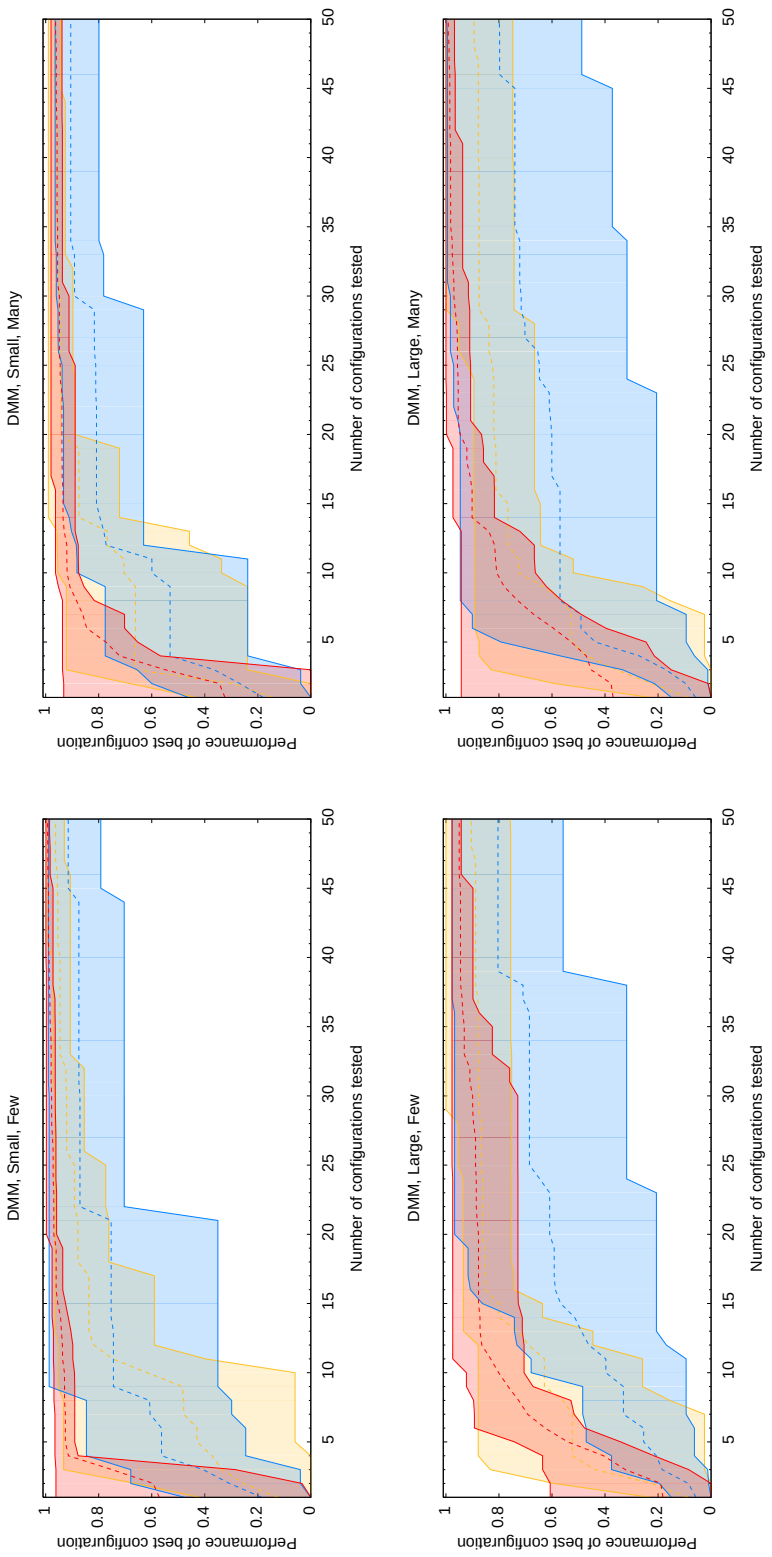


Figure A.3: Quality of best configuration found as a function of number of tests for the dense matrix multiplication application.

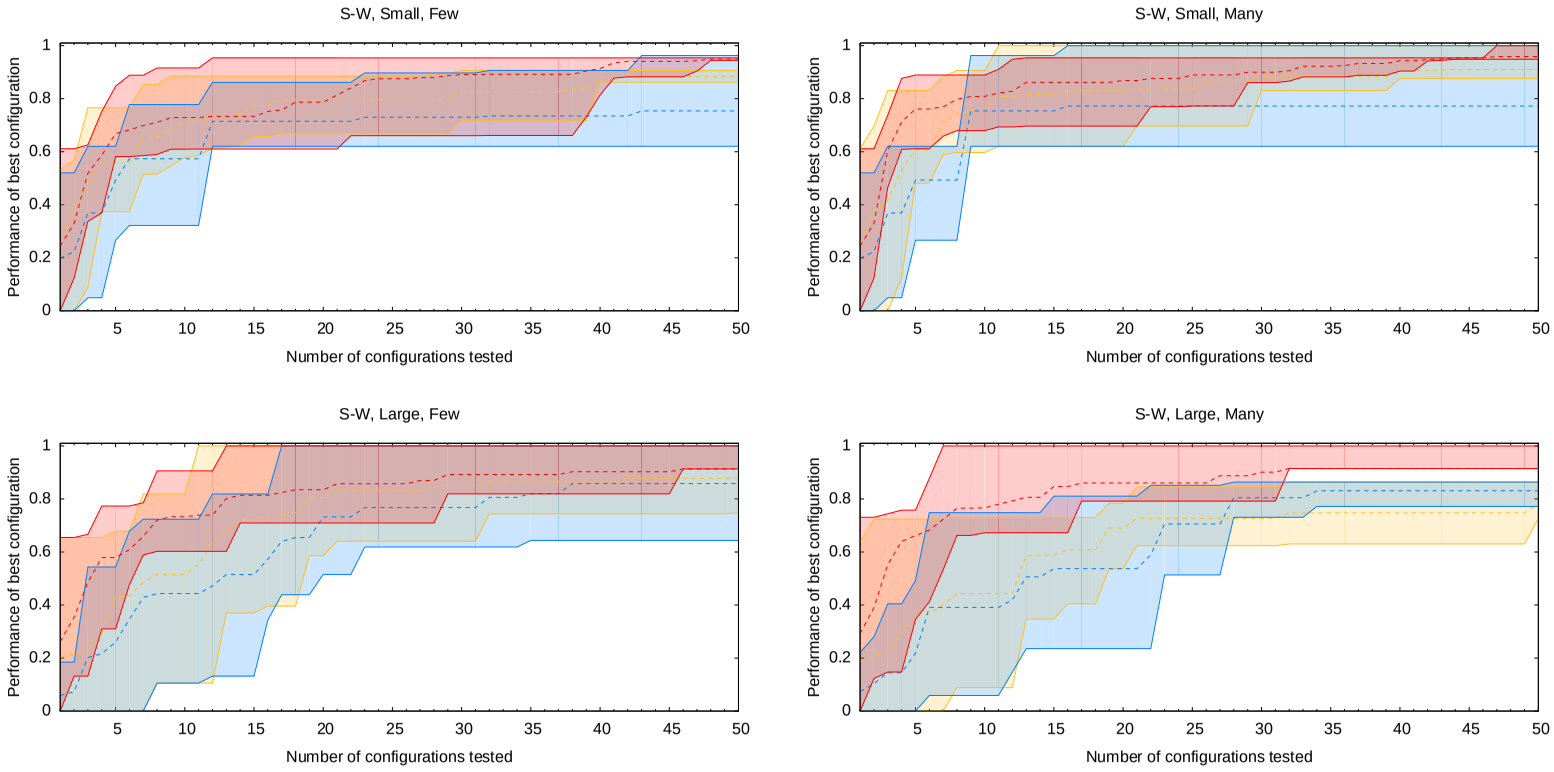
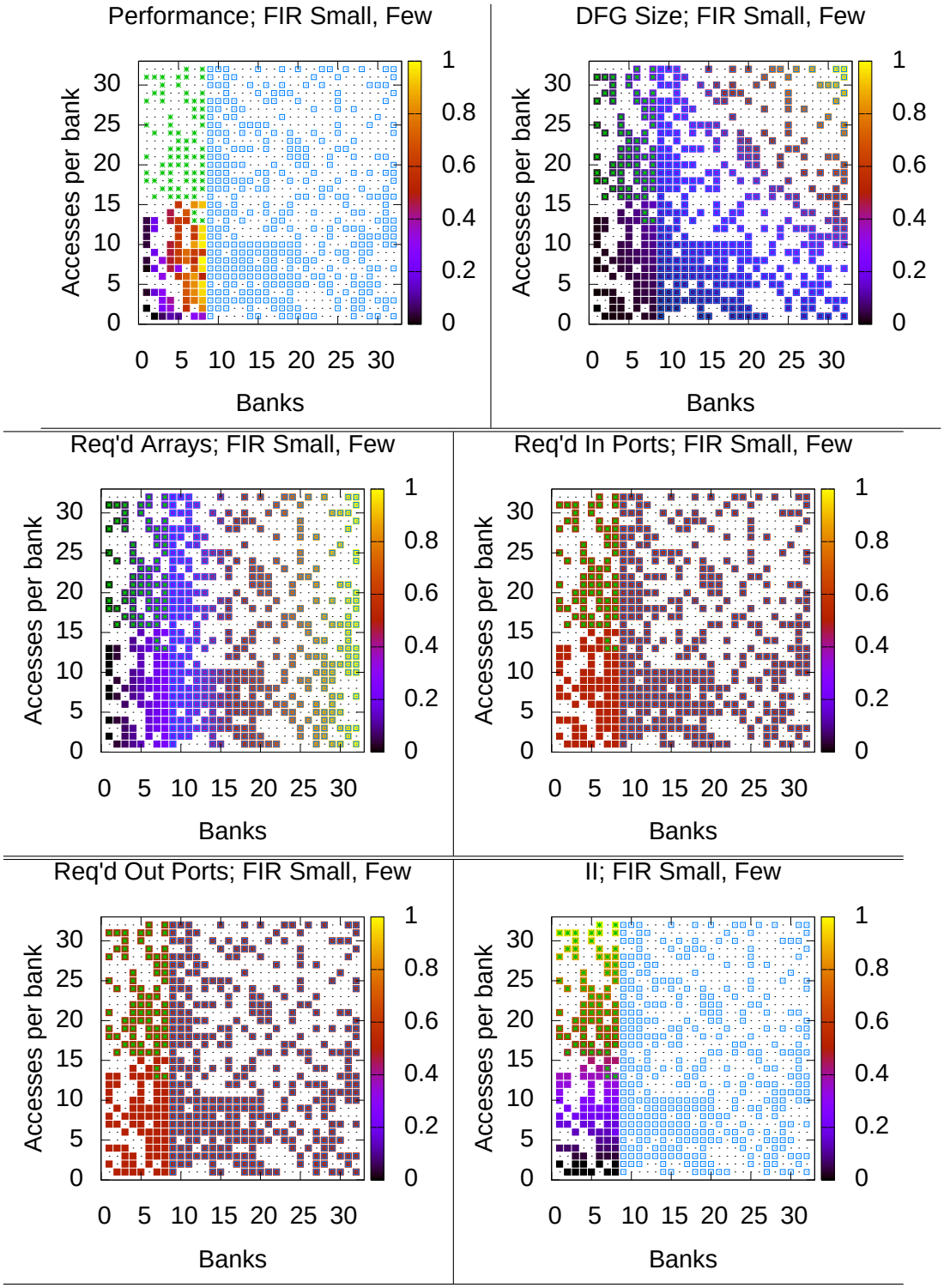
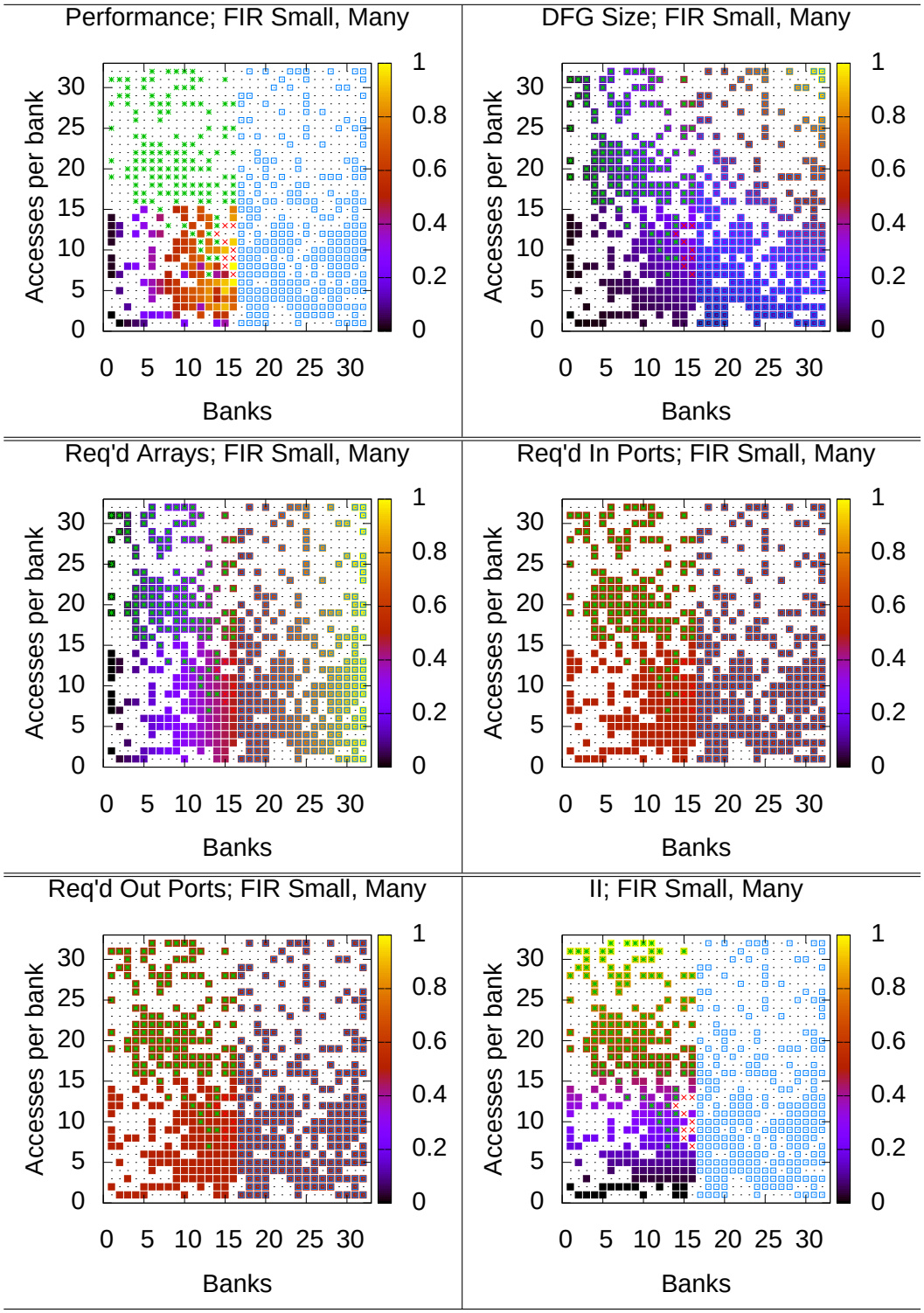
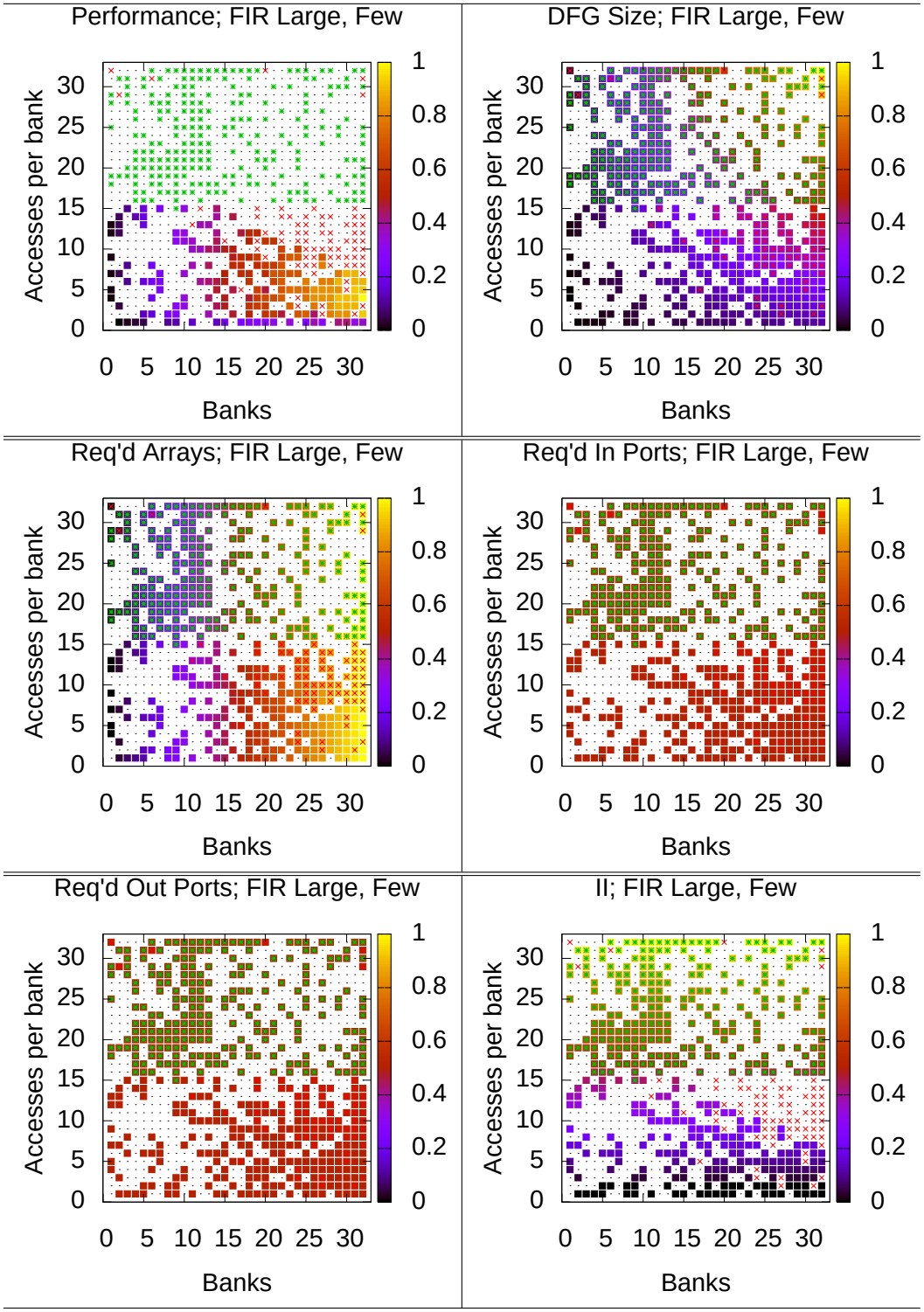
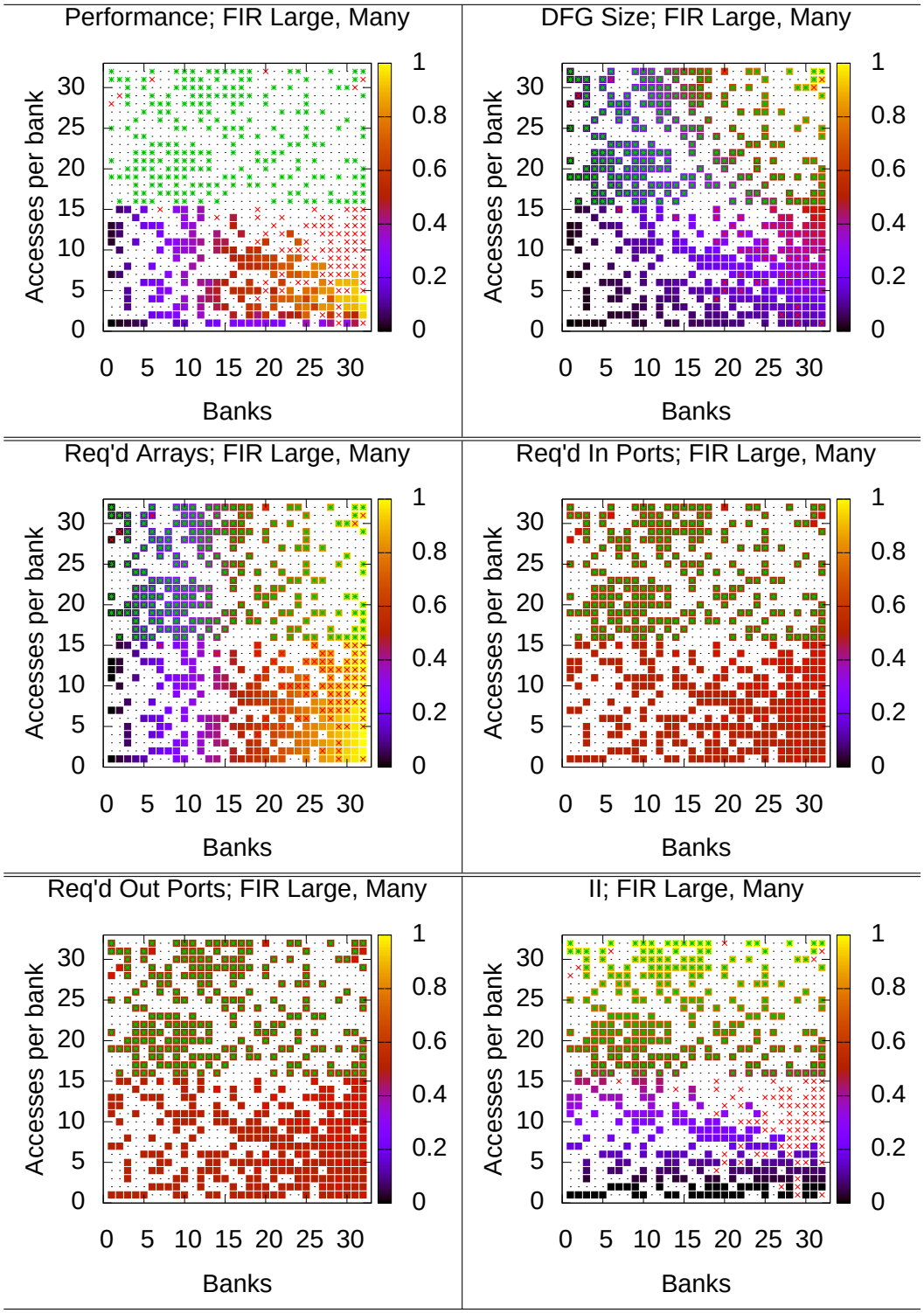


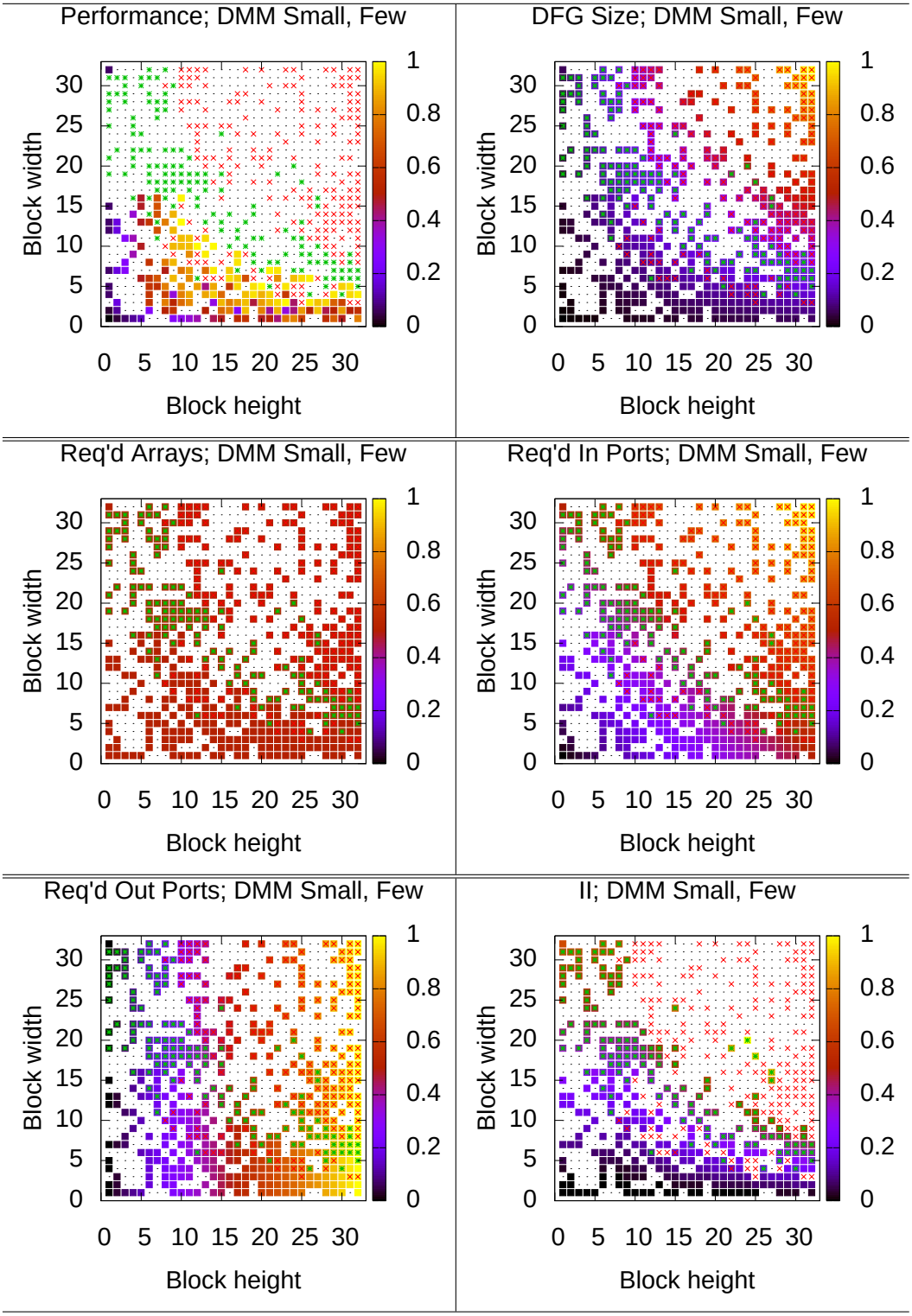
Figure A.4: Quality of best configuration found as a function of number of tests for the Smith-Waterman application.

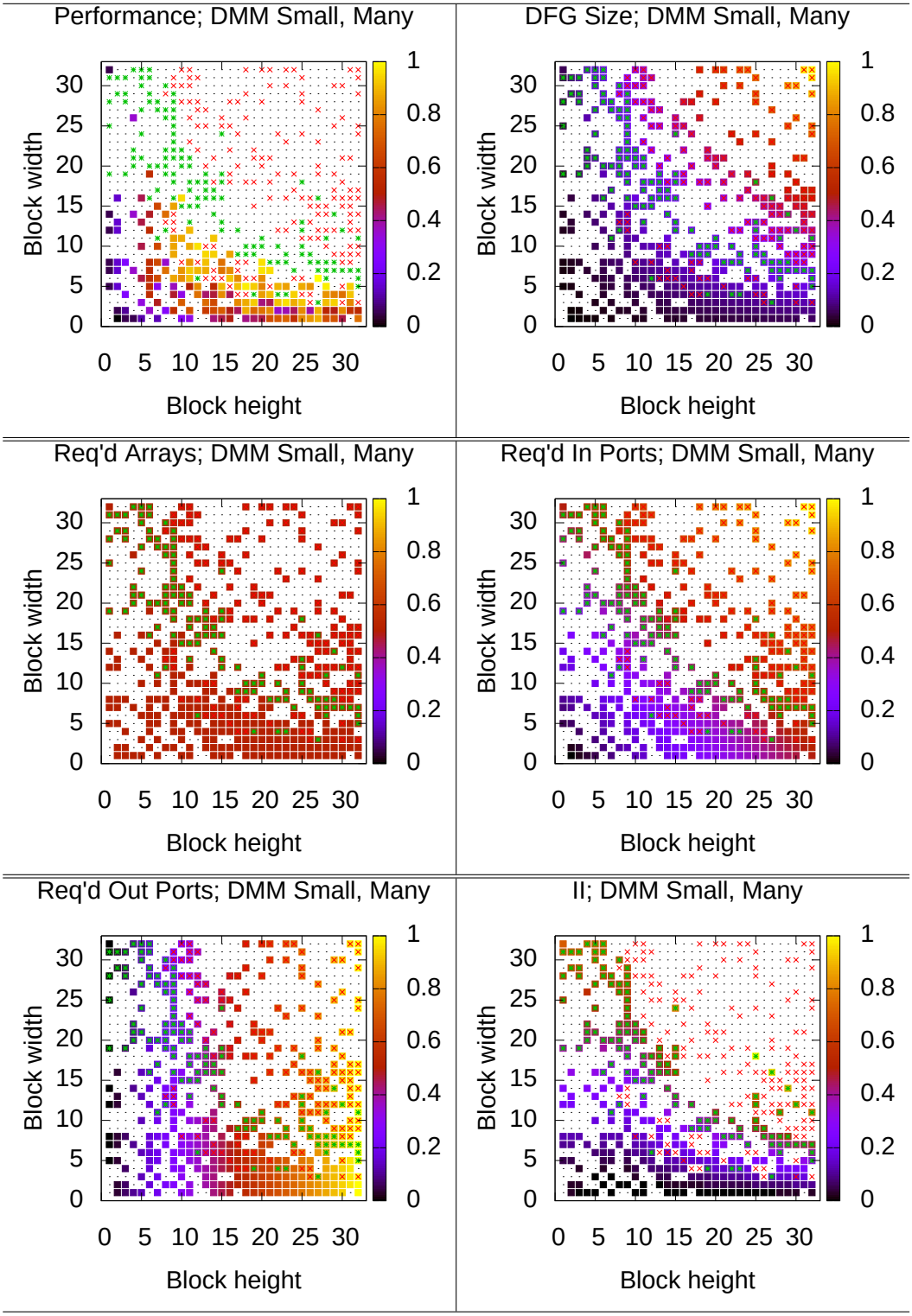

Performance; FIR Small, Few
DFG Size; FIR Small, Few
Req'd Arrays; FIR Small, Few
Req'd In Ports; FIR Small, Few
Req'd Out Ports; FIR Small, Few
II; FIR Small, Few
Accesses per bank
Banks
0 5 10 15 20 25 30
0 0.2 0.4 0.6 0.8 1

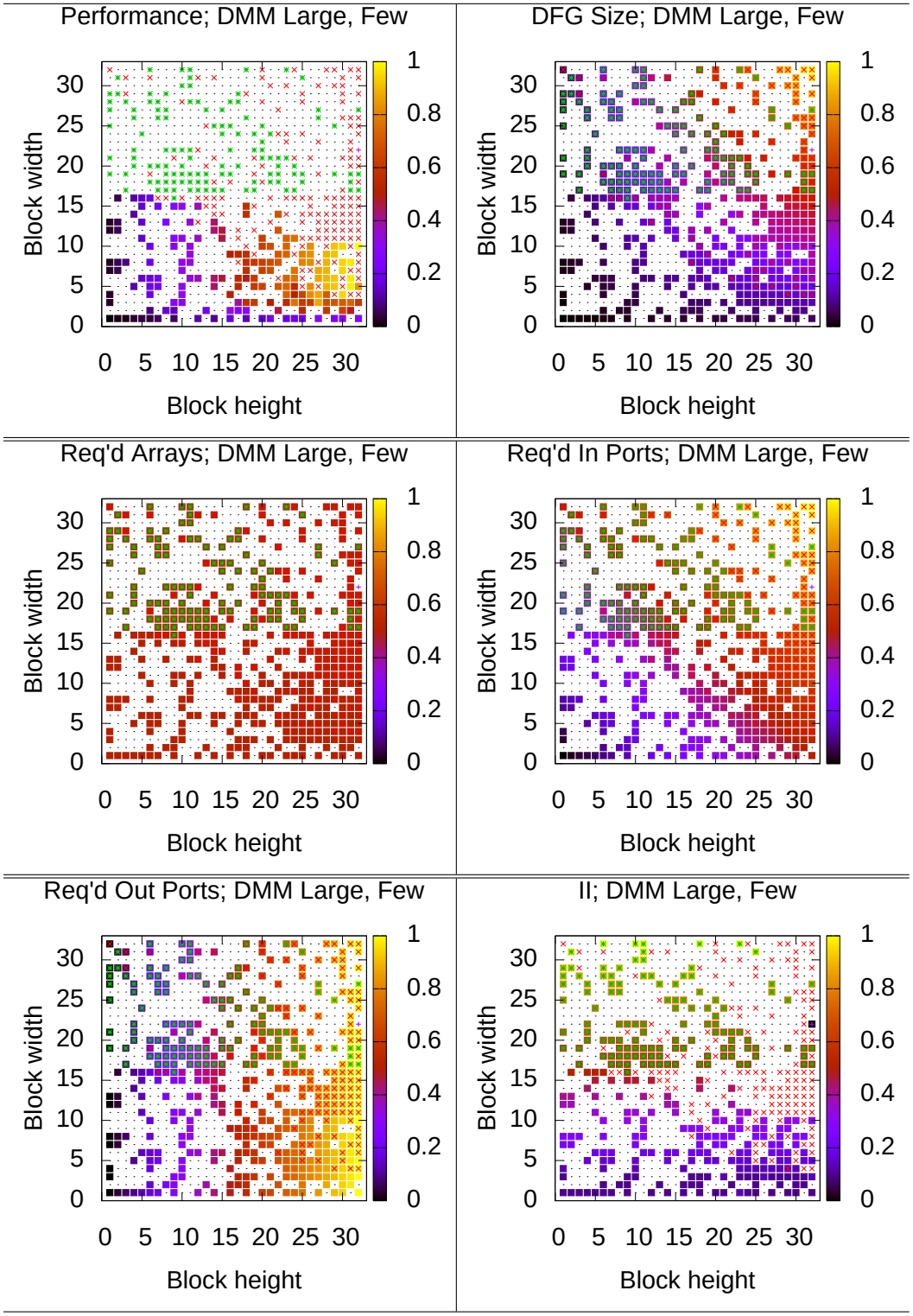

Performance; FIR Small, Many
DFG Size; FIR Small, Many
Req'd Arrays; FIR Small, Many
Req'd In Ports; FIR Small, Many
Req'd Out Ports; FIR Small, Many
II; FIR Small, Many
Accesses per bank
Banks
0 5 10 15 20 25 30
0 0.2 0.4 0.6 0.8 1

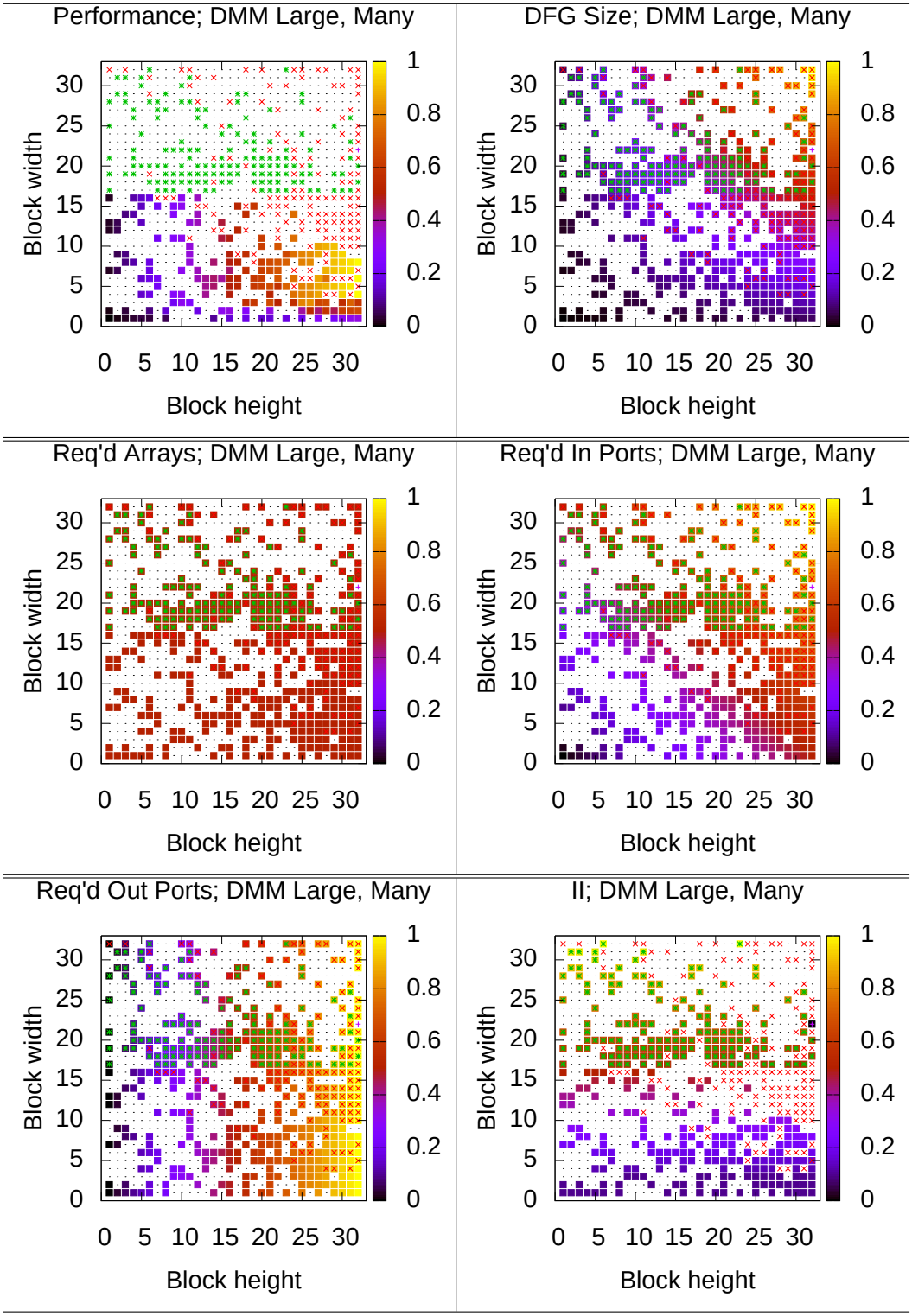

Performance; FIR Large, Few
DFG Size; FIR Large, Few
Req'd Arrays; FIR Large, Few
Req'd In Ports; FIR Large, Few
Req'd Out Ports; FIR Large, Few
II; FIR Large, Few
Accesses per bank
Banks
0 5 10 15 20 25 30
0 0.2 0.4 0.6 0.8 1

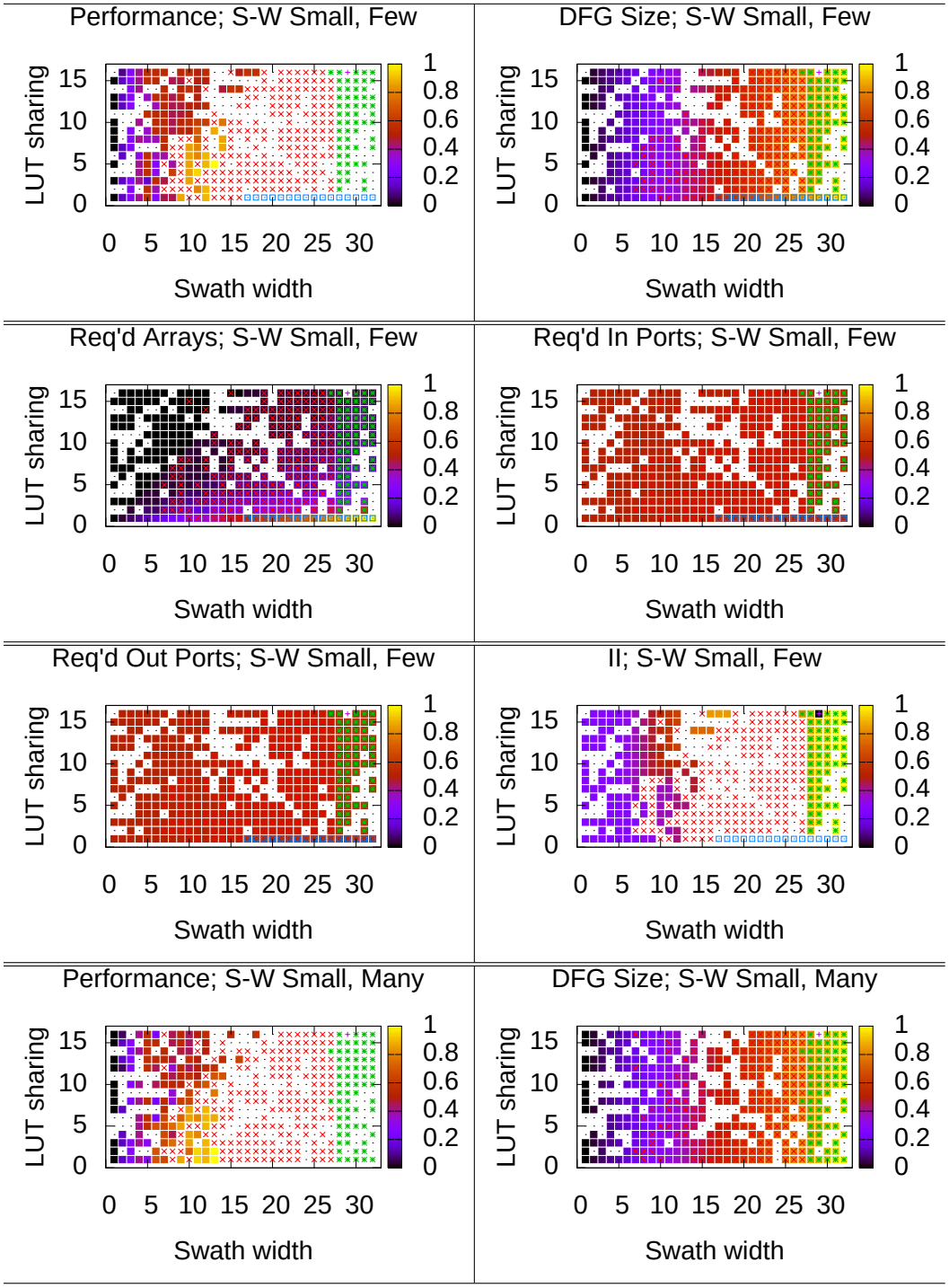

Performance; FIR Large, Many
DFG Size; FIR Large, Many
Req'd Arrays; FIR Large, Many
Req'd In Ports; FIR Large, Many
Req'd Out Ports; FIR Large, Many
II; FIR Large, Many
Accesses per bank
Banks
0 5 10 15 20 25 30
0 0.2 0.4 0.6 0.8 1

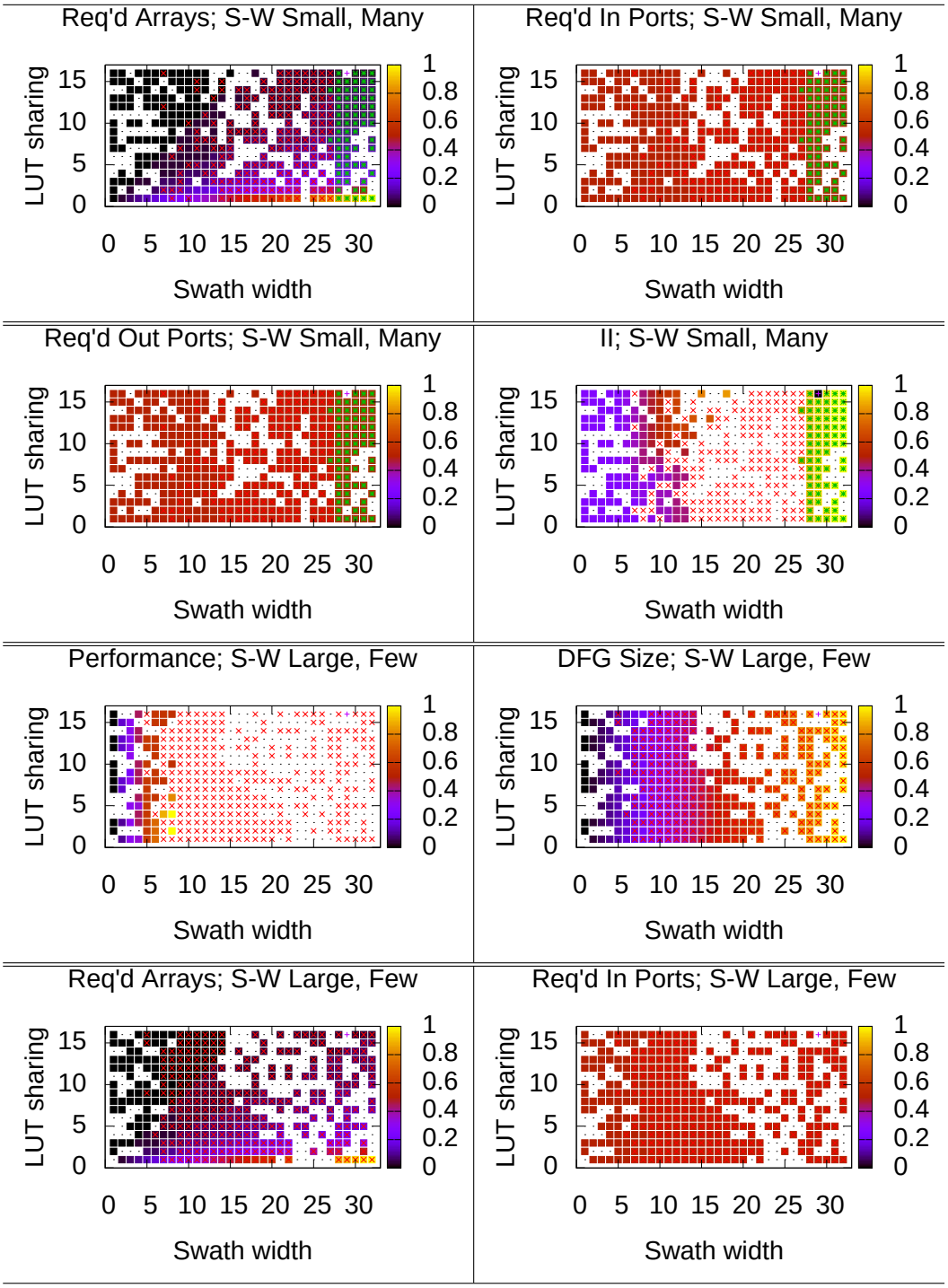

Performance; DMM Small, Few
DFG Size; DMM Small, Few
Req'd Arrays; DMM Small, Few
Req'd In Ports; DMM Small, Few
Req'd Out Ports; DMM Small, Few
II; DMM Small, Few
Block width
Block height
0 5 10 15 20 25 30
0 0.2 0.4 0.6 0.8 1

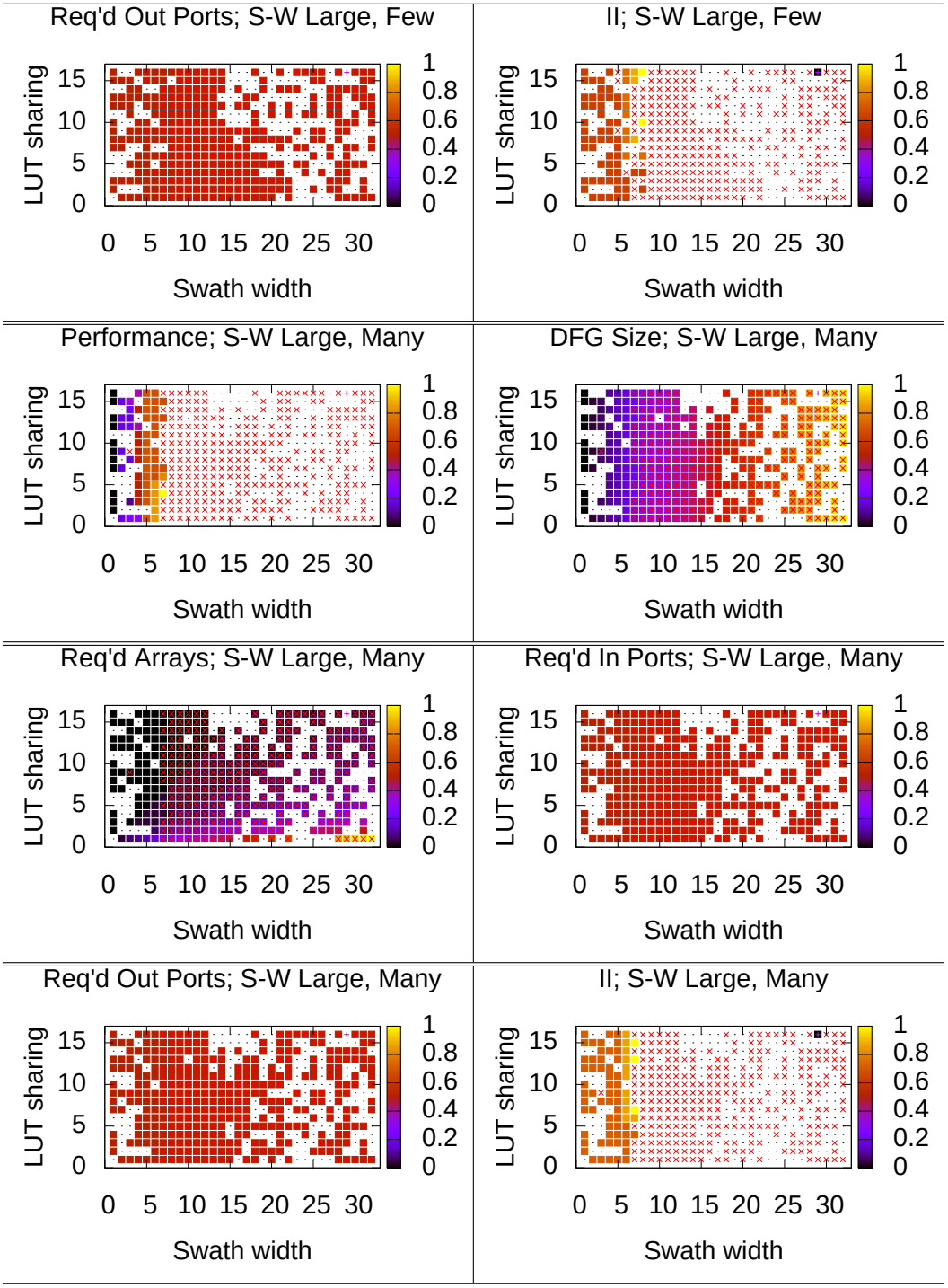

Performance; DMM Small, Many
DFG Size; DMM Small, Many
Req'd Arrays; DMM Small, Many
Req'd In Ports; DMM Small, Many
Req'd Out Ports; DMM Small, Many
II; DMM Small, Many
Block width
Block height
0 5 10 15 20 25 30
0 0.2 0.4 0.6 0.8 1

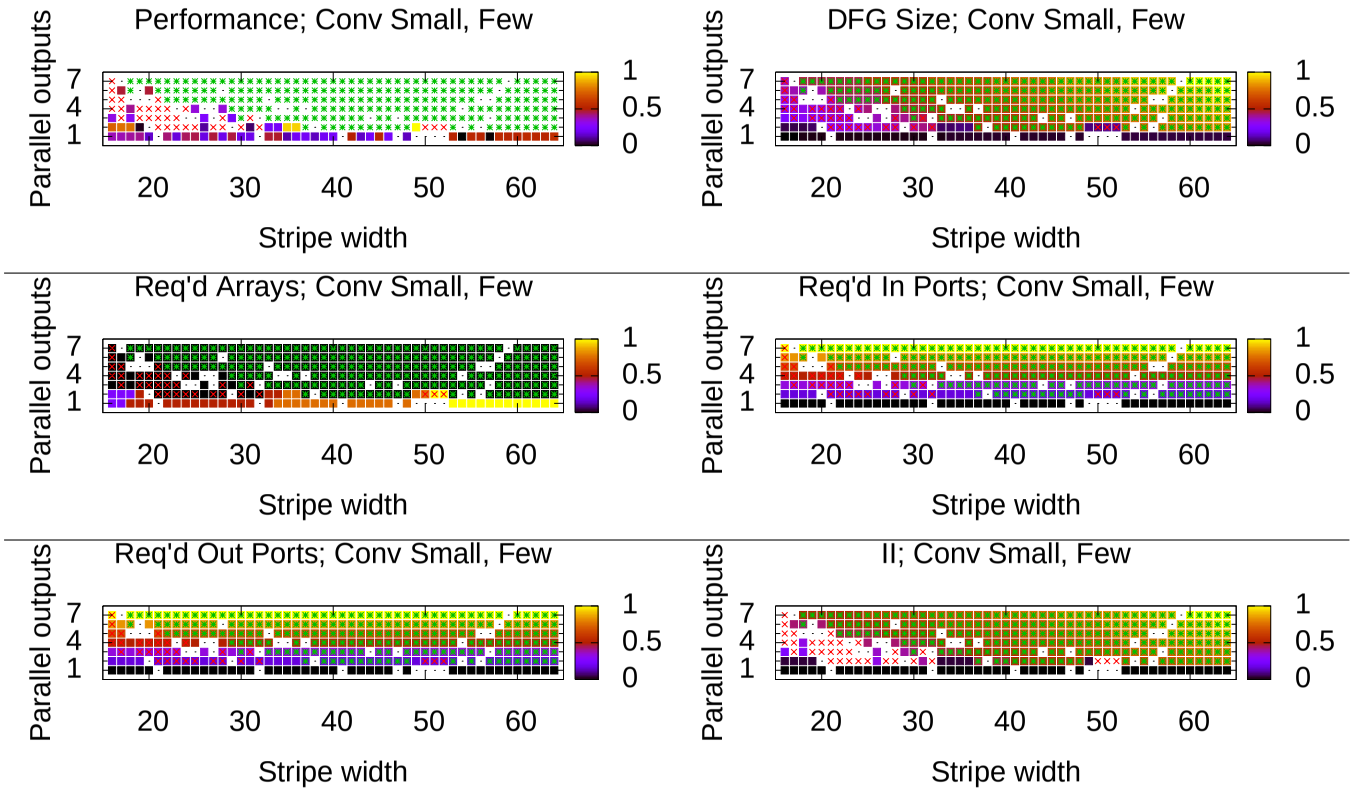

Performance; DMM Large, Few
DFG Size; DMM Large, Few
Req'd Arrays; DMM Large, Few
Req'd In Ports; DMM Large, Few
Req'd Out Ports; DMM Large, Few
II; DMM Large, Few
Block width
Block height
0 5 10 15 20 25 30
0 0.2 0.4 0.6 0.8 1

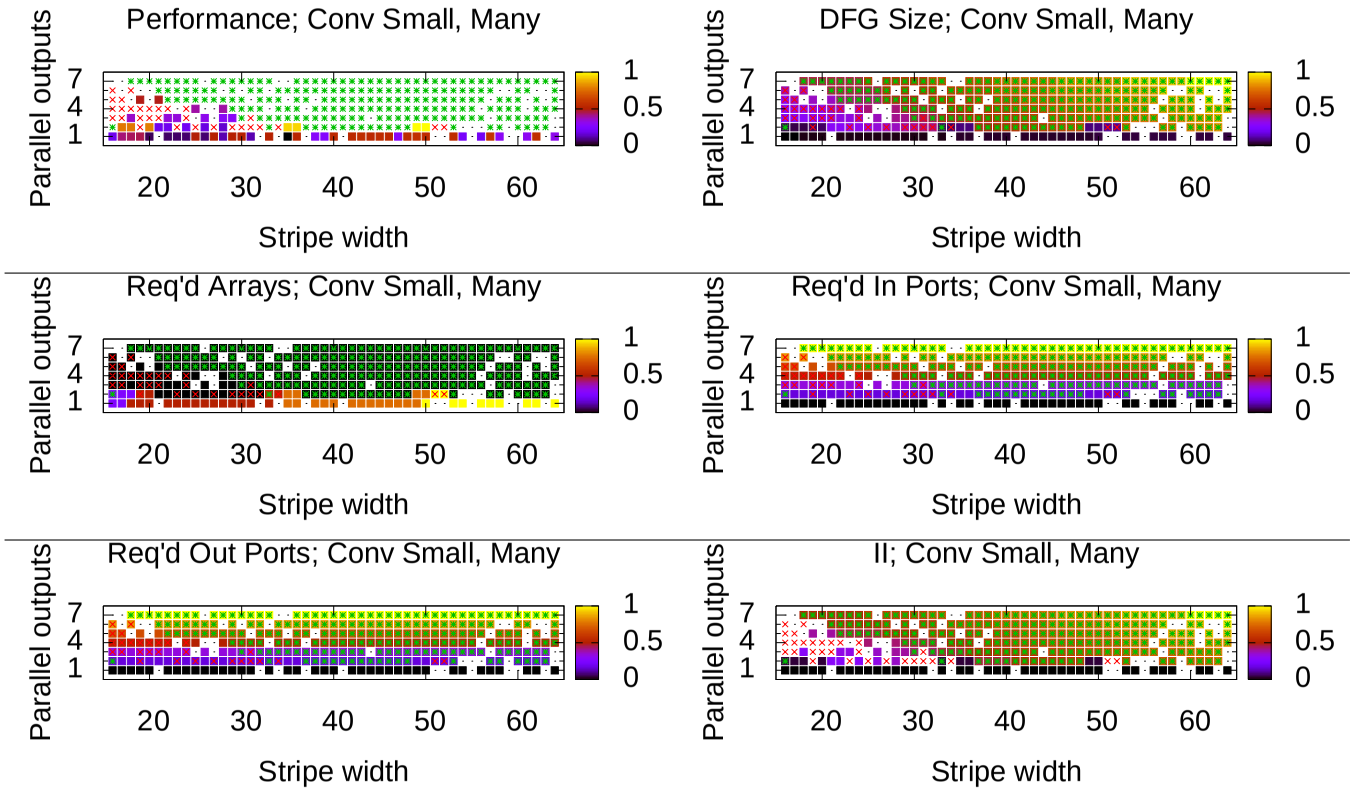

Performance; DMM Large, Many
DFG Size; DMM Large, Many
Req'd Arrays; DMM Large, Many
Req'd In Ports; DMM Large, Many
Req'd Out Ports; DMM Large, Many
II; DMM Large, Many
Block width
Block height
0 5 10 15 20 25 30
0 0.2 0.4 0.6 0.8 1

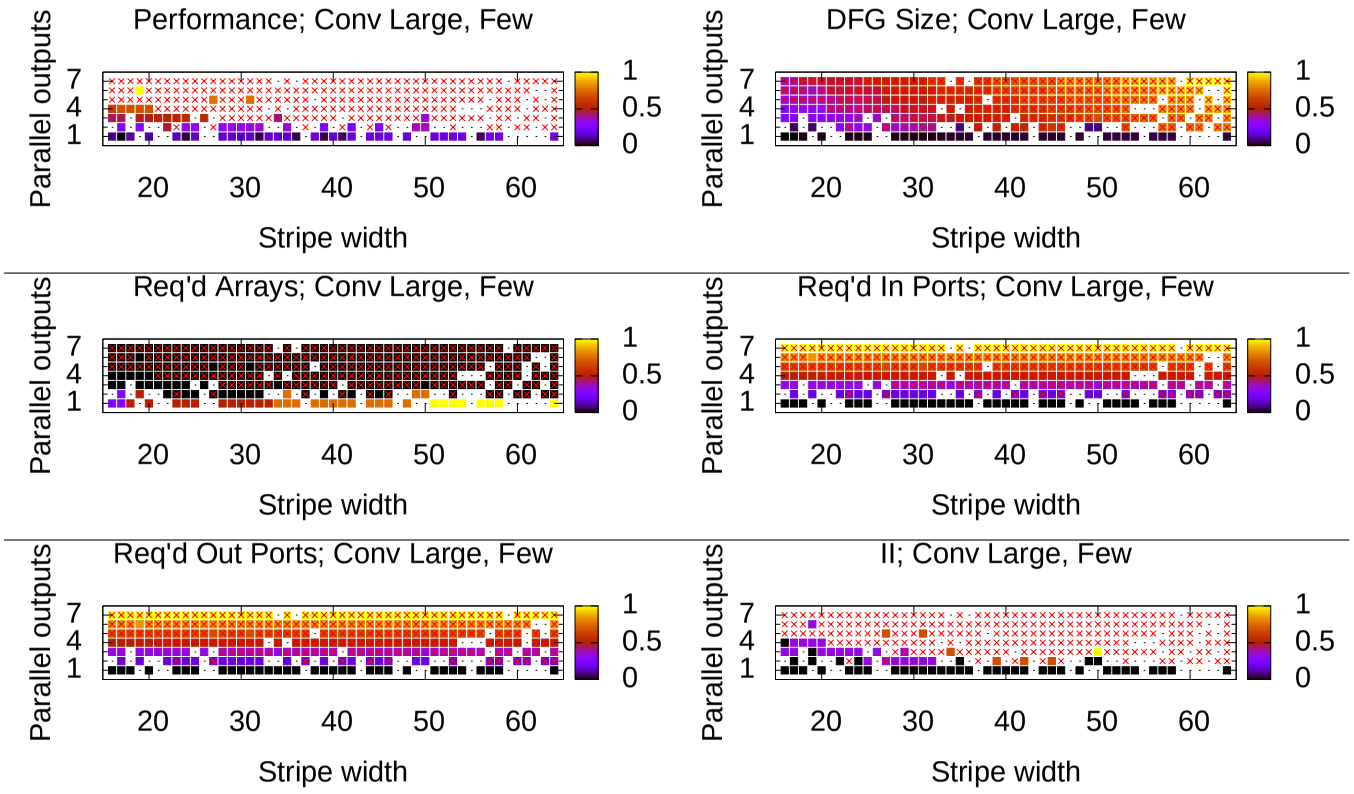

Performance; S-W Small, Few
DFG Size; S-W Small, Few
Req'd Arrays; S-W Small, Few
Req'd In Ports; S-W Small, Few
Req'd Out Ports; S-W Small, Few
II; S-W Small, Few
Performance; S-W Small, Many
DFG Size; S-W Small, Many
LUT sharing
Swath width
0 5 10 15 20 25 30
0 5 10 15
0 0.2 0.4 0.6 0.8 1

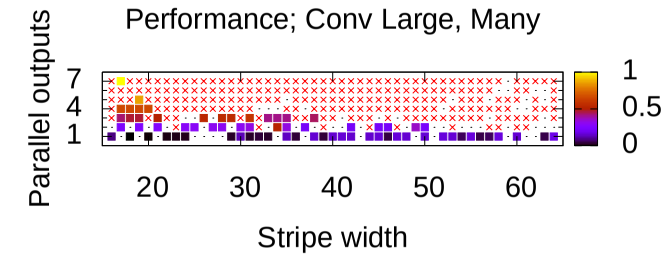

Req'd Arrays; S-W Small, Many
Req'd In Ports; S-W Small, Many
Req'd Out Ports; S-W Small, Many
II; S-W Small, Many
Performance; S-W Large, Few
DFG Size; S-W Large, Few
Req'd Arrays; S-W Large, Few
Req'd In Ports; S-W Large, Few
LUT sharing
Swath width
0 5 10 15
0 5 10 15 20 25 30
1 0.8 0.6 0.4 0.2 0

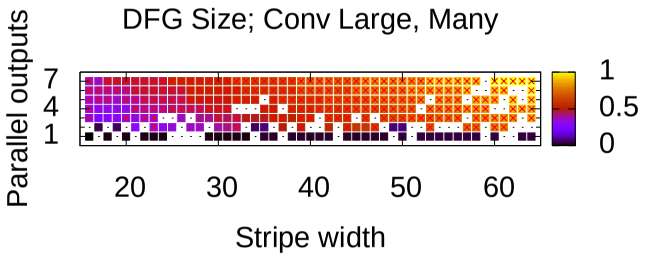

Req'd Out Ports; S-W Large, Few
II; S-W Large, Few
Performance; S-W Large, Many
DFG Size; S-W Large, Many
Req'd Arrays; S-W Large, Many
Req'd In Ports; S-W Large, Many
Req'd Out Ports; S-W Large, Many
II; S-W Large, Many
LUT sharing
Swath width
0 5 10 15 20 25 30
0 5 10 15
0 0.2 0.4 0.6 0.8 1

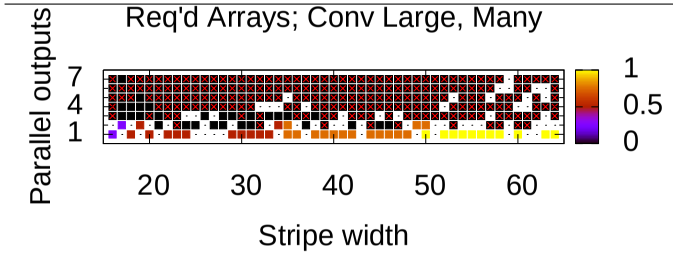

Performance; Conv Small, Few
DFG Size; Conv Small, Few
Req'd Arrays; Conv Small, Few
Req'd In Ports; Conv Small, Few
Req'd Out Ports; Conv Small, Few
II; Conv Small, Few
Parallel outputs
7
4
1
20
30
40
50
60
Stripe width
1
0.5
0

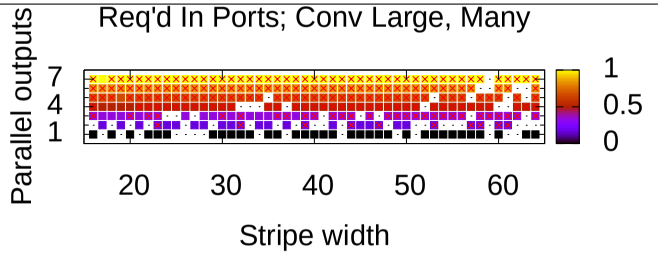

Performance; Conv Small, Many
DFG Size; Conv Small, Many
Req'd Arrays; Conv Small, Many
Req'd In Ports; Conv Small, Many
Req'd Out Ports; Conv Small, Many
II; Conv Small, Many
Parallel outputs
7
4
1
20
30
40
50
60
Stripe width
1
0.5
0

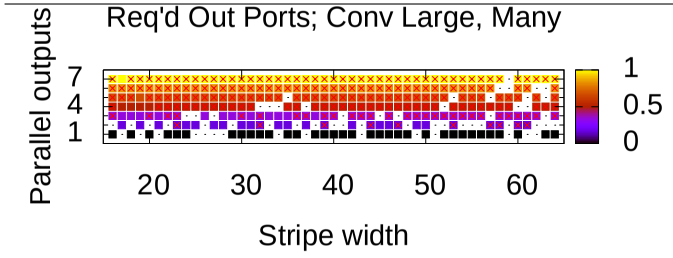

Performance; Conv Large, Few
DFG Size; Conv Large, Few
Req'd Arrays; Conv Large, Few
Req'd In Ports; Conv Large, Few
Req'd Out Ports; Conv Large, Few
II; Conv Large, Few
Parallel outputs
7
4
1
20
30
40
50
60
Stripe width
1
0.5
0

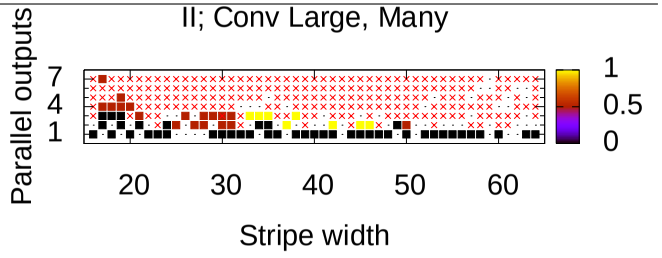

Performance; Conv Large, Many
Parallel outputs
7
4
1
20
30
40
50
60
Stripe width
1
0.5
0


DFG Size; Conv Large, Many
Parallel outputs
7
4
1
20
30
40
50
60
Stripe width
1
0.5
0


Req'd Arrays; Conv Large, Many
Parallel outputs
7
4
1
20
30
40
50
60
Stripe width
1
0.5
0


Req'd In Ports; Conv Large, Many
Parallel outputs
7
4
1
20
30
40
50
60
Stripe width
1
0.5
0


Req'd Out Ports; Conv Large, Many
Parallel outputs
7
4
1
20
30
40
50
60
Stripe width
1
0.5
0


II; Conv Large, Many
Parallel outputs
7
4
1
20
30
40
50
60
Stripe width
1
0.5
0