Architecture and Compiler Support for a VLIW Execution Model on a Coarse-Grained Reconfigurable Array

Nathaniel McVicar

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

Master of Science in Electrical Engineering

University of Washington

2011

Program Authorized to Offer Degree:
Department of Electrical Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a master's thesis by

Nathaniel McVicar

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.

Committee Members:

_____
Scott A. Hauck

_____
Carl Ebeling

Date:_____

In presenting this thesis in partial fulfillment of the requirements for a master's degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this thesis is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Any other reproduction for any purposes or by any means shall not be allowed without my written permission.

Signature _____

Date _____

University of Washington

Abstract

Architecture and Compiler Support for a VLIW Execution Model on a Coarse-Grained Reconfigurable Array

Nathaniel McVicar

Chair of the Supervisory Committee:
Professor Scott A. Hauck
Electrical Engineering

Architectures that expose parallelism to the user or the tools, such as Massively Parallel Processor Arrays (MPPAs), Coarse-grained Reconfigurable Arrays (CGRAs), and Field-programmable Gate Arrays (FPGAs), are popular, in part, due to their efficiency for solving highly data-parallel problems. Unfortunately, the more parallelism these architectures expose, the less efficient they tend to be for control heavy or serial workloads. This work proposes an alternative architecture and tool flow for the Mosaic CGRA. This flow, based on the Trimaran compiler, introduces minor hardware modifications that allow CGRA clusters to act as simple single-issue in-order processors. In processor mode, the performance per issue slot of a control-heavy kernel can be improved by as much as 4.7 times.

Table of Contents

List of Figures

List of Tables

1. Background

1.1 Highly Parallel Architectures

Field Programmable Gate Arrays (FPGAs) are currently among the most widely adopted programmable logic devices, comprising an estimated market of $4 billion in 2010 [Manners10] and $5.6 billion by 2014 [Infiniti Research Limited11]. Part of the popularity of FPGAs stems from the fact that they are powerful devices that can often come within an order of magnitude of ASIC performance, without the high initial cost. However, FPGAs perform logical operations using reconfigurable LookUp Tables (LUTs) with single bit outputs. In other words, they perform all operations at the bit-level despite the fact that most applications do their heavy lifting using 32 or 64-bit word data-paths. This mismatch leads to an inherent inefficiency in FPGAs, particularly in terms of power consumed [Liang08].

Modern FPGAs make use of dedicated word-wide arithmetic units (Altera's Variable-Precision DSP Blocks [Altera11] and Xilinx's DSP48E1 slices [Xilinx11]) to alleviate this problem to a degree, but the number of these units is very small compared to the total size of the FPGA. Instead, a number of other highly parallel architectures that perform primarily word-wide operations have gained significant popularity in recent years. These include Coarse Grained Reconfigurable Arrays (CGRAs), Massively Parallel Processor Arrays (MPPAs) and even General Purpose Graphics Processing Units (GPGPUs).

CGRAs are typically made up of a large number of word based functional units (FUs), each capable of performing basic ALU operations [Singh00]. Functional units will often contain small local memories (such as register files) and larger distributed memories may also be available, as in FPGAs. The interconnect network that allows CGRA components to communicate is also word based. CGRAs may contain some one-bit structures, such as those required to support predicated operations, but the fact that they make use of dedicated word-level hardware throughout generally makes them more power efficient than FPGAs [Barat03].

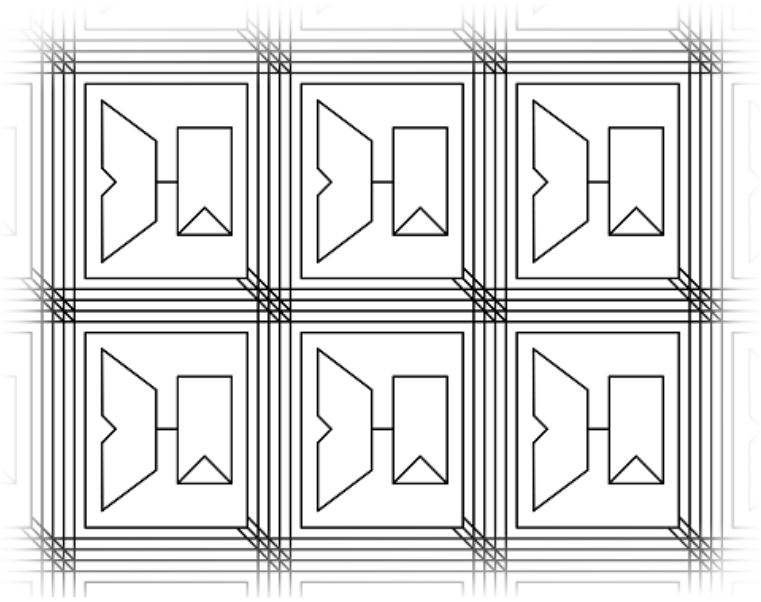


Figure 1. Generic CGRA architecture with a mesh interconnect

Although CGRAs do not suffer from the bit-level inefficiency of FPGAs, their performance is still limited in some cases by the scheduling techniques they use. Instead of employing the flexible schedule of a processor, CGRA architectures frequently use scheduling techniques where a resource has a fixed function for a given time slice independent of the input data. In contrast with this, MPPAs are constructed from a large number of simple general purpose processor cores, sometimes called processing elements (PEs). The PEs typically communicate using a routing network over which they can send data to each other. This makes it very easy to map individual components of an application to each processor on the MPPA, but it can also make it more difficult to exploit the application's inherent parallelism across the many cores.

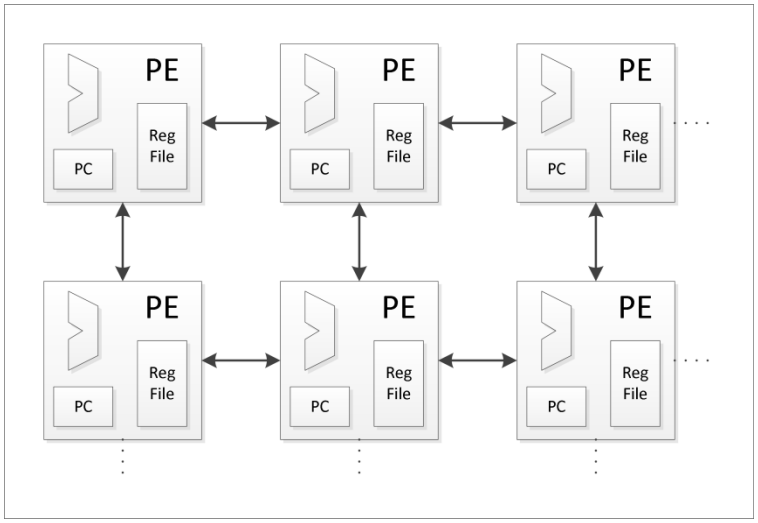


Figure 2. MPPA architecture

1.1.1 CGRA and MPPA Execution Models

As shown by Figure 1 and Figure 2, the architecture of CGRAs and MPPAs are fairly similar at a macro level. The most significant difference between the two lies in the execution models that they employ. CGRA configuration is somewhat similar to that of an FPGA, in that each functional unit and interconnect contains a configuration memory. However, unlike FPGAs, CGRAs are frequently time multiplexed. This means that each FU can be configured to perform different operations at different times. To prevent the FUs from becoming excessively complex, a fixed schedule is typically used. One example of this technique is a modulo schedule in which the same sequence of operations is executed in the same order repeatedly until the CGRA is reprogrammed. The length of the schedule, before it repeats, is called the Initiation Interval (II).

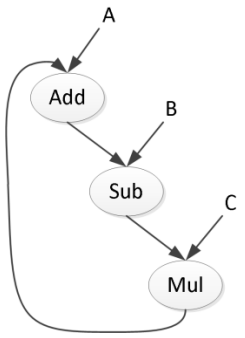


Figure 3. A simple example program

The simple program in Figure 3, which consists of a loop that adds, subtracts and multiplies, could be mapped to a single functional unit. The arrows in this figure represent data flow, and indirectly dependencies. The subtraction operation makes use of the result of the add, so it must be run after the add. The discussion of pipelining below will make the significance of this clearer. Ignoring I/O, this graph would result in an II of 3, where the functional unit adds on the first cycle, subtracts on the second cycle and multiplies on the third cycle. Through this use of time multiplexing, a single resource is able to perform three different operations without any control logic.

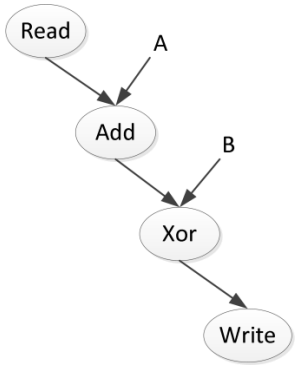
Since the modulo schedule doesn't allow for any data-dependent changes to the operations performed during runtime, CGRAs often support predication. When an operation is predicated, it will only execute if the predicate operand is true. Otherwise, the operation will behave like a no-op. Operations are also provided to set predicate bits based on comparisons. In this way, operations can be conditionally executed despite the compile-time fixed schedule, as in the following example:

```
need_jump <= !condition
jump to loop_end: if need_jump
add a, b
loop_end:
```

becomes:

```
predicate <= condition
(if predicate) add a, b
(if ~predicate) nop
```

Notice that the predicated execution example doesn't require any expensive branch operations, and the same set of instructions are executed whether or not the condition is true.



Phase	FU 1	FU 2	Iteration
0	read [0]		0
1	add [0]		
0	read [1]	xor [0]	1
1	add [1]	write [0]	
0	read [2]	xor [1]	2
1	add [2]	write [1]	

Figure 4. Example of pipelining for two FUs with an II of 2

Because the schedule of every functional unit is known at compile time, it is relatively easy for CGRA tools to exploit the parallelism in an application by spreading it across many FUs. The tools can pipeline execution across many FUs as well. Figure 4 is an example of pipelined execution. Here, some data must be read and then used for an add and xor operations before being written back. Assuming no dependencies between iterations of the loop, and two available functional units, the first one can perform the read and the add. At this point, the result of the add is ready for the xor operation. If this operation and the subsequent write are performed on the second functional unit, the first can begin a new read at the same time, as in cycle 2. Notice that the latency to complete an entire loop is still four cycles, but once the pipeline is full, a new read can be performed every two cycles. Through pipelining, in this example, double the performance of a single FU was achieved using double the hardware. Unfortunately, for workloads such as those with many branches or long sections of code that run infrequently, predication can lead to very slow execution and pipelining will may not be able to hide the delay. In these cases, the number of no-ops can outnumber the useful instructions.

In contrast to CGRA functional units, the processing elements of MPPAs typically share many of the characteristics of general purpose processors. This includes the ability to execute sequential code, stored in an instruction memory, that may contain branches. This execution model requires a program counter (PC) to store the address of the instruction currently executing. Although this model is very powerful, it does have some limitations. First, branches tend to be costly both in terms of the time they take to execute and the hardware resources required to help them execute more efficiently.

More importantly, it is much more difficult for a compiler to create code to execute in parallel on many MPPA cores. In the CGRA, many FUs running on a modulo schedule of the same length can work together to execute a large number of arithmetic or other operations, as described above. Through pipelining these operations can easily make use of data from a different FU, and the temporal relationship can be statically determined by the compiler. Unfortunately, MPPA cores cannot be as tightly coupled. Since data-dependent branches make it difficult for the compiler to predict the relationship between the program counters of two separate PEs, MPPAs have to rely on more heavyweight inter-core communication mechanisms such as message passing.

For a concrete example of this issue, consider the small loop from Figure 4. On a CGRA using modulo scheduling it was possible to achieve a speedup of 2x by using two FUs. However, due to communication penalties this would not be possible on a basic MPPA, since a send operation would be required to move data from one PE to the other. A receive operation would then be required at the second PE to get the data. This would result in the ability to accept new data every three cycles once pipelined, instead of every two cycles as was possible on the CGRA. Additionally, the MPPA implementation would be slowed down further by the time required for the branch operation at the end of each loop iteration. Some more advanced MPPA architectures, such as Ambric [Butts07], provide features to get around these limitations. Ambric provides zero overhead loop operations, and is able to send results out to the communication fabric as part of a regular ALU operation. In the case of an MPPA with those capabilities, the performance advantage will go to the architecture whose tools can extract the most parallelism. In these situations the simpler CGRA schedule becomes a significant advantage.

1.2 Mosaic 1.0 Architecture

Mosaic 1.0 is a CGRA developed by the labs of Scott Hauck and Carl Ebeling at the University of Washington [UW Embedded Research Group, 2006]. Mosaic is made up of a heterogeneous array of Functional Units connected by a configurable interconnect network (Figure 5). Each group of FUs and surrounding logic is called a cluster, and cluster elements are connected with a crossbar. This allows each functional unit within a cluster to communicate with any other without significant restriction, although only a

limited set of signals in the FUs are connected to the crossbar. The FUs in a cluster share a single connection to the routing network through a switchbox. This switchbox is also connected to the intra-cluster crossbar, and the routing network between Mosaic 1.0 clusters is made up of a combination of statically and dynamically scheduled channels [Van Essen09].

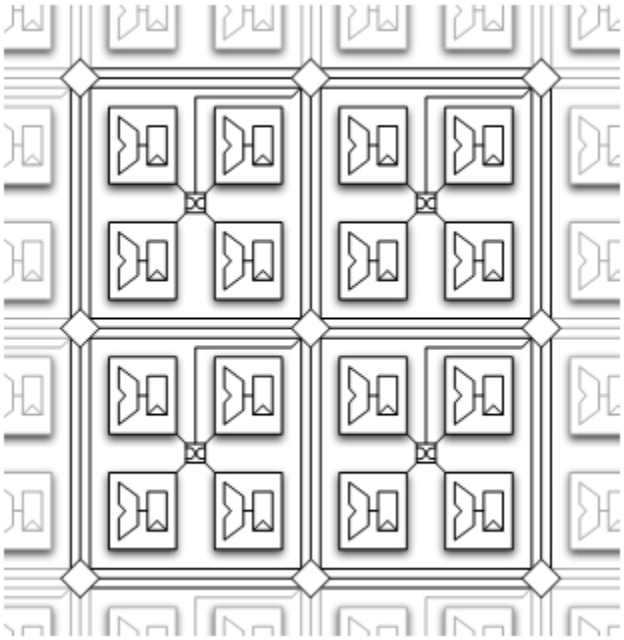


Figure 5. Mosaic 1.0 architecture. [Van Essen10]

The configuration for each component of a Mosaic cluster is stored in SRAM, in a manner similar to an FPGA. However, unlike an FPGA, there are II configurations for each element, instead of just a single one. This SRAM requirement puts a limit on the maximum II the architecture supports, probably 128 in the case of Mosaic 1.0.

In addition to the Functional Units, each cluster contains local memory, one or more large rotating register files or retiming chains, as well as some additional small register resources. As mentioned, these resources are connected by a 32-bit corssbar. The Mosaic Functional Unit itself can perform basic ALU operations as well as shifts. Some Functional Units are capable of multiplication, but only integer arithmetic is supported. In addition to these 32-bit resources, there is a 1-bit interconnect to the FUs. This resource is used for predicate bits and also includes LUTs and register files. The specific architecture design that will be used in this document, going forward, features four FUs per cluster

and large rotating register files. This architecture is summarized in Figure 6 and described in detail in [Van Essen10].

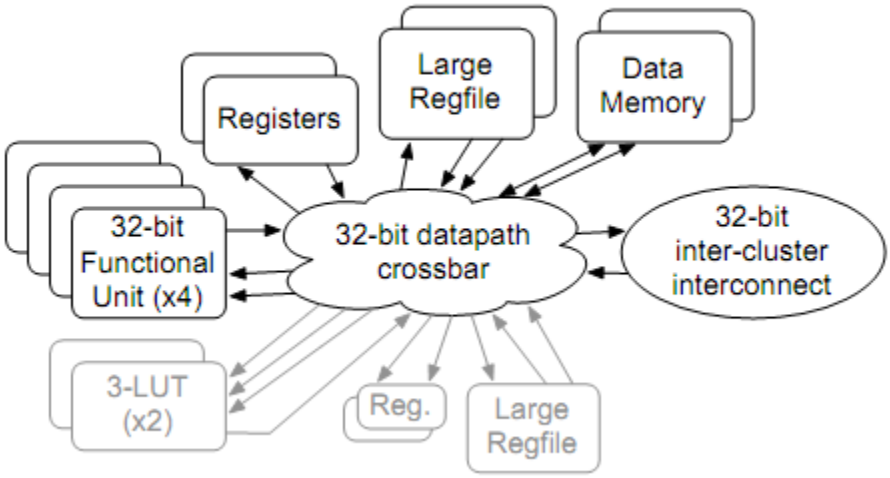


Figure 6. Mosaic 1.0 Cluster. [Van Essen10]

One limitation of Mosaic 1.0 is that it only supports execution of a single kernel at a time. This will be described in more detail in the next section, but most importantly all clusters operate in lockstep. Additionally, every cluster shares the same logical modulo counter, so each is loading from the same slot in configuration SRAM. One consequence of this mode of execution is that when a blocking operation that can't be completed occurs, such as a read from an empty stream or a write to a full one, all clusters must stall their execution.

1.3 Mosaic 1.0 Toolchain

To understand the motivation for the work presented in this Thesis, as well as Mosaic 2.0, it is important to understand the Mosaic 1.0 toolchain. An overview of this toolchain is provided in Figure 7. The user provides an application, written in the C-like Macah language, and an architecture specification. The Macah compiler compiles the application to a Verilog description, and the Electric architecture generation tool translates the architecture specification into an input that Schedule, Place and Route (SPR) can process. Using these two inputs, SPR spreads the program across the architecture, creating a configuration for Mosaic. This specifies the II of the kernel and the state of each

component during every cycle in the schedule. This configuration can be simulated in the PostSim tool, where correctness can be verified and power modeled. The configuration could also serve other purposes, including programming a physical CGRA device.

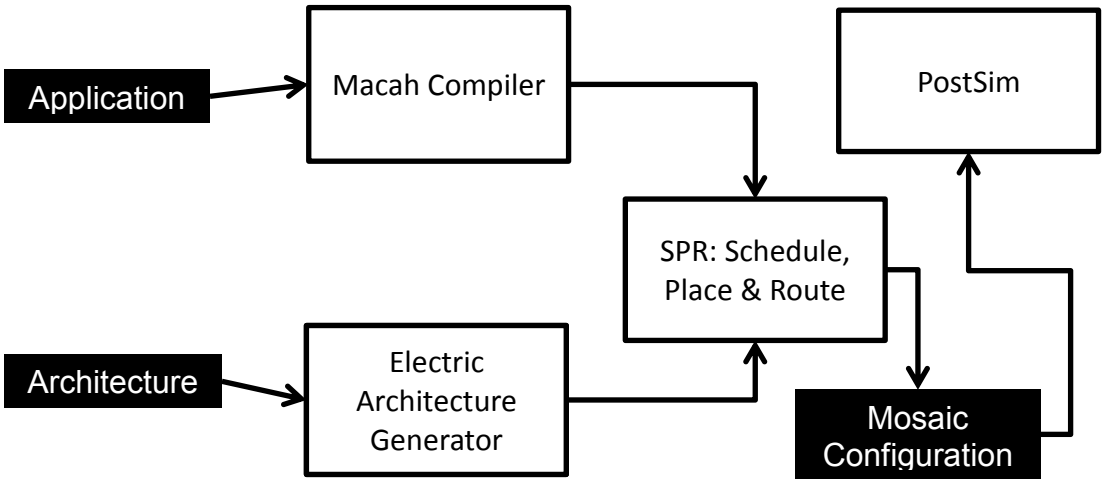


Figure 7. Mosaic 1.0 Toolchain

1.3.1 Macah

Macah is a programming language with syntax and features very similar to C [Ylvisaker08]. However, Macah also contains additional features that make it well suited to streaming applications targeting a CGRA like Mosaic. The properties of the CGRA also place restrictions on the language features used in the portions of the Macah code that is intended to run on the CGRA.

A top level Macah 2.0 function is substantially different than the `main` function of a C program. Instead of containing sequential program code, it consists of a configuration block that wraps one or more task blocks. The configuration can use loops and other control logic to construct tasks, but this configuration logic cannot be evaluated at runtime. All tasks are constructed at the beginning of the run of Macah program, referred to as configuration time.

A Macah task is where all of the code executed at runtime is contained. There can be many tasks in a configuration block, and they are all executed in parallel. The primary restriction on general tasks is that no task can access the same variables or memory as

another task. This eliminates a lot of the programming challenges faced by traditional concurrent applications.

Tasks communicate using a streaming paradigm. Streams are written in one task and read in a different task, so they provide the only way to pass data from one task to another. Macah streams can be created using any data type, although, as mentioned above, Mosaic doesn't support floating point arithmetic. Both input and output streams provide blocking and non-blocking operations. The blocking read or write will stall the task until its operation can be completed. This will happen in the case of a full stream buffer on a write, or an empty stream buffer on a read. In the case of a non-blocking stream operation, the operation will complete regardless and return a Boolean value indicating if the operation succeeded or not. Macah stream reads and writes are performed using special receive and send operators respectively (Table 1). For the non-blocking case, `op` is a Boolean value, which will be set to true if a value was read or written, and false if a blocking operation would have stalled.

Operation	Receive	Send	Non-block Receive	Non-block Send
Operator	`var <? strm`	`strm <! var`	`op :: var <? strm`	`op :: var <? strm`

Table 1. Macah stream operators

Not all Macah tasks can be run on the CGRA. Instead, only tasks containing a kernel block can be accelerated. These tasks must meet additional restrictions to make them suitable for CGRA execution. First, although normal tasks can only communicate through streams, they can access external memory as long as no other tasks access that memory. Kernel tasks may only access memories declared inside the task. In practice these memories must be small enough to get mapped to memory blocks or registers inside the CGRA. All memory allocation in a kernel task must also be static. In addition to memory restrictions, kernels can't make calls to functions. The exception to this is the case where the function can be inlined.

Finally, Macah has a few language features to make it easier to write deeply pipelined kernels with a low II. The first of these is the `FOR` loop. The use of an uppercase `FOR` loop instructs Macah to unroll the loop. This allows separate loop iterations to be executed in parallel, assuming that they don't have any dependencies from one iteration

to the next. Macah also allows the declaration of shiftable arrays. These arrays function just like regular C arrays, but they introduce array shift operators. The array right and left shift operators adjust the array indices by the amount shifted. For example, if an array is right shifted by two, the value originally accessed by `array[3]` is now located at `array[5]`. Values that are shifted in from outside the bounds of the array are undefined. Shiftable arrays are very useful for accelerator applications, due to their ability to make programs more regular by allowing a given instruction to always access the same array index. They also map well to rotating register files where the array shift can be implemented as a register file rotation.

1.3.2 SPR

Schedule, Place and Route (SPR) is the tool that maps a compiled Macah program to a specific Mosaic architecture [Friedman09]. A detailed discussion of SPR is outside of the scope of this thesis. It is sufficient to note that SPR is the tool which is primarily responsible for mapping the parallelism available in the program onto the CGRA. This includes assigning operations to resources both in space (selecting hardware to perform the operation) and time (choosing an issue slot). The current version of SPR is designed for Mosaic 1.0, and can only perform these tasks for a single kernel.

1.4 Mosaic 2.0

Mosaic 2.0 is the next evolution in the Mosaic CGRA architecture. It is designed to address one of the most significant shortcomings of Mosaic 1.0, the fact that all clusters operate in lock-step on a single kernel. This is a significant limitation for two reasons. First, the II of the entire kernel is limited to the longest recurrence path anywhere in the kernel. Second, all stalls are global, so any stream that is full or empty will bring the entire CGRA to a halt.

Mosaic 2.0 addresses these issues by supporting multiple kernels executing on the CGRA simultaneously. To get an idea of how useful this can be, consider an application such as the one in Figure 8. This application consists of a filter kernel followed by a computation kernel. Each kernel has a single input stream and a single output stream. The filter kernel has a recurrence II of 3, and although it can accept a new input every iteration it filters

most of them out, producing a new output only every tenth iteration on average. The computation kernel performs a complex computation which can't be pipelined as effectively. It has an II of 5 and in the steady state can accept one input and produce one output every iteration.

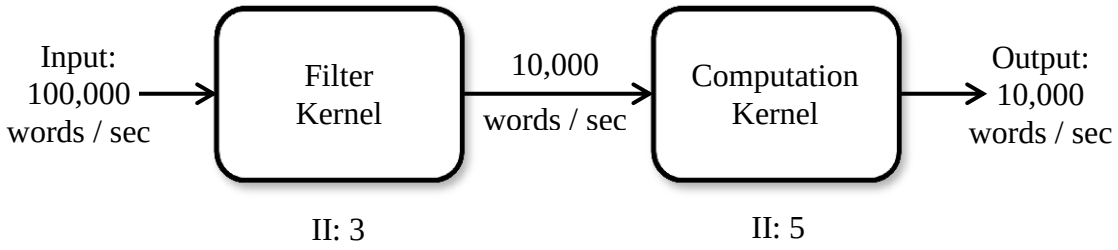


Figure 8. Example Mosaic 2.0 Application

For Mosaic 1.0, this entire application would share a single modulo counter, which would be forced to conform to the longer II of 5. This means that the filter portion of the kernel would only be able to accept a new input every five cycles, since it requires at least three cycles to process a single input. This will reduce performance by almost half. In the case where there was a temporary shortage of data on the input stream or an output FIFO was full, both parts of the application would be forced to stall. In some cases this is not efficient. For example, without this constraint in the situation where the computation kernel has a full output FIFO, the filter would be able to continue accepting new input until its own output buffer was full. Supporting simultaneous execution of multiple kernels, as in Mosaic 2.0, resolves both of these problems.

The primary feature of Mosaic 2.0 is allowing each of many kernels on the CGRA to have their own modulo counter. Each of these modulo counters is be able to stall independently. Additionally, Mosaic 2.0 could potentially support configuration of fabric resources based on predicates, allowing for more jump-like behavior and alleviating some of the restrictions of the CGRA execution model [Friedman11]. There are other minor complications as well. Neighboring clusters in the same kernel can also receive delayed versions of their neighbors' modulo counters, and the clusters are connected using the interconnect described in [Panda11].

With more kernels come more challenges for the programmer. Although infinite buffers on streams can be modeled in a simulator, real buffers are finite. This introduces the potential for kernels that are well behaved in the average case to deadlock under some loads. For instance, kernel A sorts incoming values into two streams, which both go to kernel B, and approximately half of the values go to each output stream. Kernel B reads one value from each stream and does some processing. This situation is illustrated in Figure 9.

Kernel A:

```
kernel {
  …
  {
     val <? in_strm;
     if (val > threshold)
        strm_1 <! val;
     else
        strm_2 <! val;
  }
}
```

Kernel B:

```
kernel {
  …
  {
     val_1 <? strm_1;
     val_2 <? strm_2;
     work(val_1, val_2);
  }
}
```

Figure 9. Potential multi-kernel deadlock situation

Typically, this will work perfectly. However, in the case where many values above the threshold arrive sequentially, `strm_1` could fill up at the same time kernel B empties `strm_2`. At this point the kernels will become deadlocked, with A stalling on the write to `strm_1` and B stalling on the read from `strm_2`. In Mosaic 2.0 avoiding these deadlocks is the programmer's responsibility, as the tools place no restrictions on the data rates between kernels.

1.5 Trimaran

The Mosaic 2.0 design described above is very tightly coupled to the modulo counter execution model. Trimaran, on the other hand, makes use of the program counter model. Trimaran is a research Very Long Instruction Word (VLIW) compiler that was developed as a collaboration between many academic and industry laboratories [Chakrapani05]. Trimaran supports a number of architectures in addition to VLIW, and most of the

properties of the target architecture can be defined using an MDES machine description file. Trimaran also includes a cycle accurate simulator for application profiling.

Figure 10 shows an overview of the Trimaran toolchain. Trimaran begins with a standard C input file, and initial parsing and standard compiler optimization are performed by OpenIMPACT. Next, the Elcor stage takes in the MDES file and the intermediate representation (IR) produced by OpenIMPACT and performs various architecture specific optimizations and compilation tasks. The output of the Elcor stage is Rebel IR. Finally, the SIMU simulator can run simulations and performance modeling on the application. The first stage of SIMU, called Codegen, takes in the Rebel IR and the MDES, and produces, among other output, a list of operations somewhat similar to an assembly file. It is important to note that this file is more abstract than a typical assembly file, and it doesn't contain important information like branch address and complete register assignments. More detailed information on any of these stages can be found in [Trimaran 07].

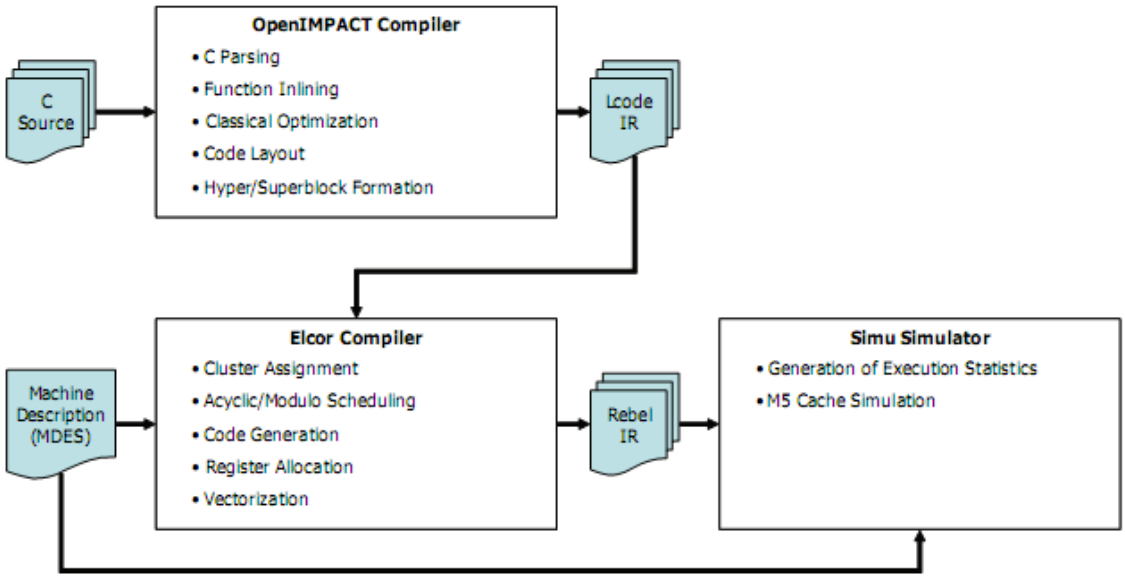


Figure 10.Trimaran System Organization

Because of its reasonable performance [Chobe01] and flexibility, the Trimaran compiler has been used in projects involving various VLIW and more traditional superscalar architecture, include IA-64 and ARM [Chakrapani01]. In addition to these more

traditional processor architectures, researchers have occasionally used Trimaran to target CGRAs [Yan11] or other similar collections of functional units [Middha02]. The modifications to Trimaran required for the Mosiac project are discussed in the next section.

2. Software Design

The engineering effort involved in this project can be divided into two components, software design being the first of these. As discussed in 1.1.1, the modulo schedule execution model has significant limitations. In cases where substantial code is executed conditionally, all of this code must still be executed on the CGRA, prior to the outputs of these code sections being ignored due to predication. This introduces inefficiencies both in power and area. Large quantities of work may be performed and not used, and the logic to do this work could consume many clusters on the CGRA. Sufficiently complex control may even be difficult for SPR to route.

All of this leads to the conclusion that it would be desirable to have some way to execute sequential code with complex control on the CGRA, even at a substantial loss in parallel performance for that specific kernel. This could be necessary in cases where a complex but low throughput kernel is part of the critical path in an operation, preventing moving the kernel's logic off-chip. In a standalone Mosaic situation, there may not even be a general purpose processor on which to run these kernels.

This software engineering effort was motivated by a desire to get a sense of the performance that could be achieved by targeting Mosaic 2.0 hardware components as a VLIW processor using Trimaran. The first step in this effort is to add support for some kernels in a Macah program to be compiled by Trimaran, instead of Macah. Next, Trimaran must produce output that is in a form able to configure Mosaic functional units instead of running through SIMU. It's important to note that, as mentioned briefly in 1.4, this is just one of multiple concurrent efforts to support less restricted code on Mosaic 2.0 in some manner.

2.1 Macah Modifications and Scripting

The first stage in compiling Macah kernels with Trimaran is designating these kernels for Trimaran compilation and getting their code to Trimaran in a usable form (standard C). This required a series of minor modifications to Macah, and some scripting.

Conveniently, Macah is capable of generating C versions of compiled Macah programs. This code is normally used for rapid correctness simulation of the Macah application. It makes use of a runtime library, containing C implementations of unique Macah features such as shiftable arrays and streams. Other Macah features, such as unrolling `FOR` loops, are handled prior to emitting the C code.

To add Macah support for Trimaran kernels, additional functions to mark the beginning and end of Trimaran kernels were added to this library and the Macah compiler. These functions do nothing in Macah simulation, but are preserved by the compiler and remain in the final C code. There, they provide hooks for the scripts that convert Macah C output to Trimaran C input.

```
configure_tasks {
  …

  task first {
    … // no kernel
  }

  task second {
    …
    kernel second {
      …
    }
  }

  task third {
    …
    trimaranKernelStart();
    kernel third {
      …
    }
    trimaranKernelEnd();
  }
}
```

Figure 11. Pseudocode For Trimaran and Macah Tasks

Figure 11 shows the usage of functions marking the beginning and end of a Trimaran kernel. When these functions are placed around the kernel as seen in the figure, the

`macahtotri` Perl script can parse the Macah output C file and create a Trimaran C file containing a standard `main()` function. The body of this function will be the contents of the kernel, including any relevant variables declared in the task, even if these variables are declared prior to the call to `trimaranKernelStart()`. No other variables are available, with the exception of streams. This isn't a significant additional limitation because kernel tasks don't have access to external variables in the Macah language.

Aside from the scope of variables, the script currently has some additional limitations. Most significantly, only a single kernel in a given Macah file can be designated as a Trimaran kernel. This is due to the fact that this kernel becomes the `main()` function for an entire Trimaran run. Under this system, multiple communicating Trimaran kernels can only be implemented by compiling them separately and combining them at configuration time. No tool that supports this sort of configuration currently exists. Similarly, the current Macah flow will still compile kernels tagged for Trimaran and run them through SPR in the same manner as any other kernel. Figure 12 shows the completed tool-chain given support for arbitrary kernel combinations. Note that the architecture specifications used are limited to the subset of architectures that support Trimaran's execution model. These will be discussed in much more depth in subsequent chapters.

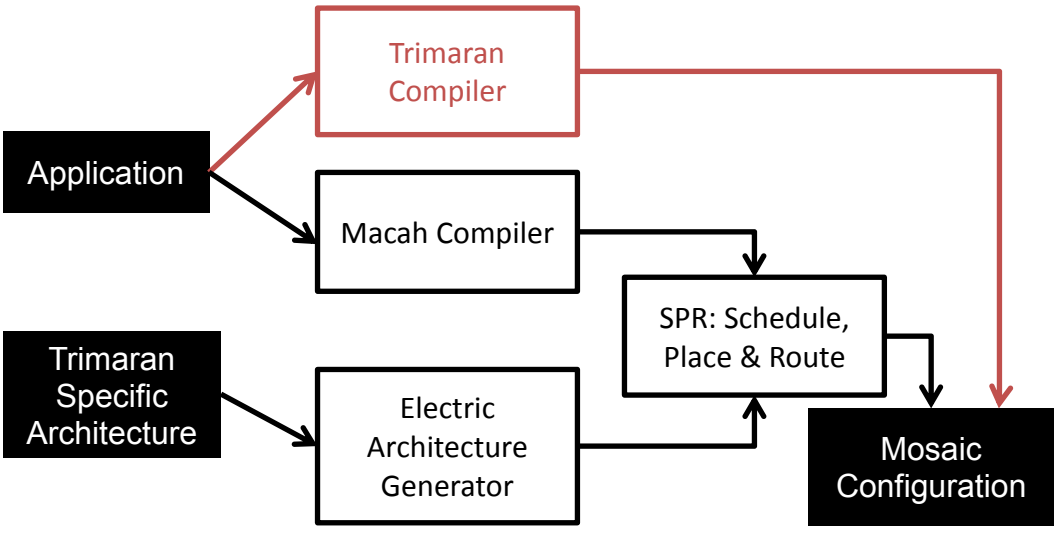


Figure 12. Macah / Trimaran Hybrid Tool Flow

In addition to this limitation, the Trimaran flow does not support any of the Macah features provided by the runtime library mentioned above, other than streams. Streams are supported through conversion to volatile variables, where reads and writes are normal assignments. After this conversion, the output of the `macahtotri` script is standard C, perfect for Trimaran compilation.

2.2 Trimaran Modifications

The Mosiac configuration shown in Figure 12 is a Verilog file, describing the state of the configuration SRAM for every architecture component at each cycle of the modulo schedule. The information in any given set of configuration bits varies based on what is being configured. It could be the operation an ALU should perform during the cycle, which input a multiplexer should select or the enable bit on a register. If a component is not doing any useful work in a given cycle, for example if the ALU output is not multiplexed to any storage location, it's value can be left unwritten for correct simulation or written to the same value as in the previous cycle to conserve power. It is important to note that the configuration slots and cycles here are the same as those used by SPR for modulo scheduling, as discussed in 1.1.1. When SPR produces a configuration, it will include the II and some value for each component in every cycle within that II. For example, if the II of the application, as mapped to the available hardware, was 3 there would be state for each architecture component for cycles 0, 1 and 2.

In the case of Trimaran compiling for a CGRA, the modulo counter acts as a more traditional PC and Trimaran must produce the correct configuration to carry out an operation for each architecture component. This is very different than the output of a traditional compiler. Normally, a compiler will produce simple instructions that are then expanded to control each architectural component during the decode stage of execution. In this case, Trimaran must produce what is essentially all of the post-decode state. Because of this, Trimaran requires much more extensive knowledge of architecture specifics than is typically required by a compiler. This issue will be discussed in more detail in the architecture sections below.

No changes are required to the initial two stages of Trimaran, OpenIMPACT and Elcor, from what was discussed in 1.5. All significant Trimaran modifications are in the final Simu stage, shown in Figure 13. Here, Trimaran provides two main mechanisms for producing output. The first is the Simu cycle accurate simulator which executes the program on the machine running Trimaran. This simulator loads and converts the program from the Rebel IR produced by Elcor. Each operation, or in some cases class of operations, is implemented in the simulator as a function taking the inputs to the operation and storing the output. These functions are implemented as a simulation library called Emulib. During this process, many detailed statistics are collected. Unfortunately, this mechanism is poorly suited to producing Verilog output. At no point during this execution in the simulator is there a time when all of the information about an operation is available. For example, the function that represents an operation accepts a value of an input register instead of which register that value came from. Similarly, register values are sometimes stored directly in variables or memory, and the mapping back to a register is not preserved.

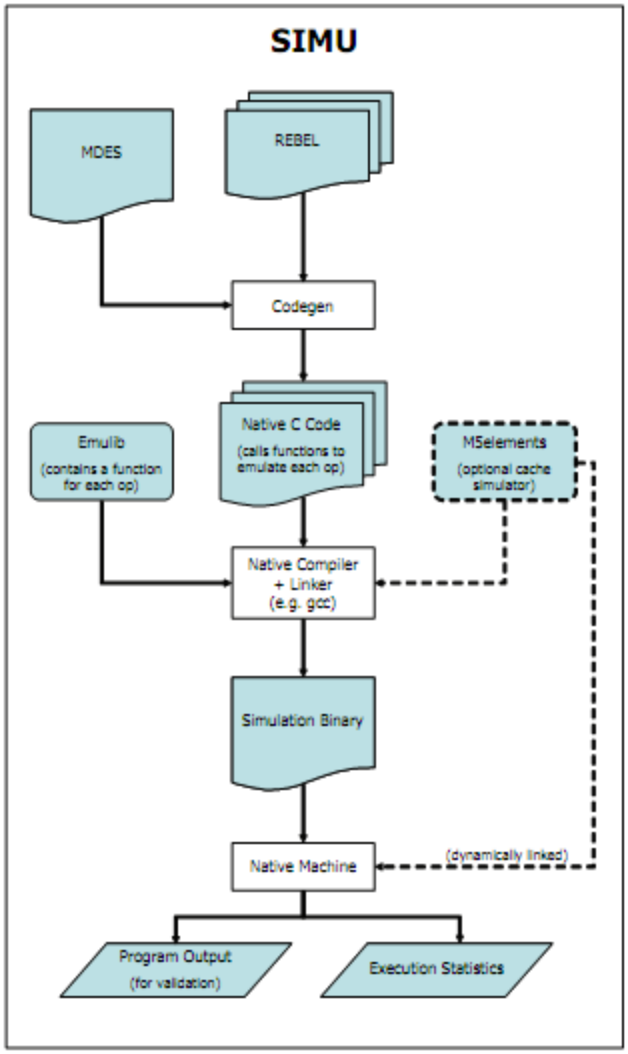


Figure 13. Simu Architecture [Trimaran 07]

The second output mechanism is much more useful for producing the required Verilog configuration files. During the Codegen stage seen in Figure 13, Trimaran can write a number of files. In addition to the native C code, that uses Emulib to execute the program being simulated, Codegen also produces a number of table files similar to the memory tables in a traditional executable. These tables map variables and functions to memory locations, and additionally in this case, include register accesses and other information for each operation. A file similar to a very high level assembly output is also produced, although this file alone is incomplete and has insufficient information about each instruction to produce the required Verilog output.

This stage of Codegen is perfect to produce the Verilog configuration file because the Rebel IR has been parsed and all of the required information is available and being written to the files described above. This information includes all of the register and constant operands required by each operation as well as the mapping of these operations to the multiple ALUs and other compute resources in the VLIW case. One of these additional resources is a rotating register file, which could rotate at the same time as an ALU operation executes during a single cycle.

The configuration generation performed in this stage is conceptually simple. Verilog configuration for each component is written based on what it might be required to do during the cycle. For example, if the operation is an add, the source register files are configured to read the source registers, the crossbar is configured to route those registers to the ALU, the ALU is configured to add and the crossbar writes the ALU output back to the register file. The configurations currently produced by Trimaran assume that all instructions can execute in a single cycle. Streams are handled as a special case, in which the load and store operations (which are always present because streams are declared as volatile variables) are replaced by accesses to dedicated stream hardware.

3. Architecture Design

A variety of architectures, designed with different goals, were produced during the course of this project. They are described in this section.

3.1 Custom Trimaran Processor

As discussed in 2.2, the Triamaran modifications that allowed for the production of Verilog output require detailed architectural knowledge to operate correctly. As a proof of concept for this aspect of the compiler, a custom architecture was produced out of the same Architecture Generator components used in [Van Essen10]. Just like these Architecture Generator components, which are used to construct the CGRA architectures processed by SPR, this architecture was specified in the Electric CAD tool [Rubin10].

Electric is an open source CAD program written in the Java language with many properties that make it well suited for the Mosaic project. Electric supports a number of levels of circuit abstraction, ranging from schematic and HDL description down to physical circuit layout. For Mosaic, individual architecture components are specified at the HDL level and then connected as schematics. Electric allows the user to write plugins that can perform complex tasks, such as connecting components in a design to a crossbar or generating an entire architecture from a simple specification. Finally, Electric can output the entire design as a Verilog architecture description that is read in by SPR.

The custom Trimaran architecture was a very simple Electric design, consisting of a large crossbar connecting each component (Figure 14). The Trimaran control registers, including the PC and counters used for low overhead loops, are implemented using simple registers in Electric. The adder, for incrementing the PC, and the other ALUs, used for loops and ALU operations, make use of the same arithmetic components that lie at the heart of a Mosaic functional unit. In the standard Trimaran architecture branch targets must always be stored in a branch target register file before the branch. This register file, along with the general purpose and predicate files, use the same Cydra rotating register file [Dehnert89] as in Mosaic. Finally, the constant generation and Stream I/O are performed by custom components designed for this processor. This was necessary because constants in Mosaic enter through register files at the beginning of

execution, and this approach would require significant compiler modification to use with Trimaran. Memory mapped stream I/O operations are performed using custom Verilog stubs that allow for test data to be sourced from and written to files during simulation.

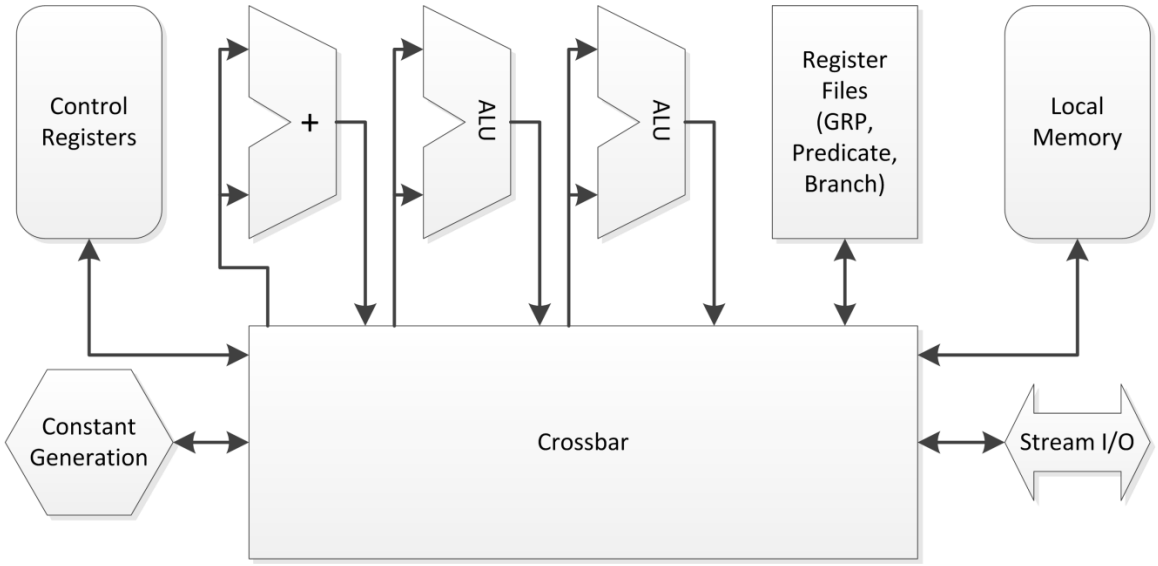


Figure 14. Simple custom Trimaran processor design

This design is clearly very different from a traditional simple single-cycle processor architecture. Connecting through a large cross-bar is extremely inefficient, given that many of the possible connections don't make sense. For example, the compiler would never generate an instruction where the output of the stream I/O goes directly to the PC or the memory writes to an ALU input. The more natural architecture would feature dedicated connections between related components, with multiplexing where necessary. The components used are also not a perfect fit. For example, some Trimaran logical and arithmetic operations require two of the Mosaic FU ALUs chained together. This would likely not be the preferred structure for a real single-cycle processor.

Both of these design decisions stem from the fact that this Trimaran-only processor is a proof of concept for compiling C code to Mosaic hardware using Trimaran. This model provides much more useful information, for example potential integration with Mosaic power and timing models, than it would if it was written in custom Verilog. The use of the crossbar is based on the fact that this more closely resembles the flexibility of the actual Mosaic architecture, so even though most of this flexibility is wasted in the

Trimaran processor, it requires a Verilog emitter much closer to what would be required to configure the Mosaic architecture.

3.2 Custom Trimaran Processor and Mosaic 1.0 Comparison

The custom Trimaran processor from the previous section includes the baseline components required for execution of Trimaran programs. A comparison between the custom processor and the current Mosaic architecture can help to determine what Mosaic components will require modification to reach a consensus architecture. As discussed above, some portions of the current Trimaran toolchain, particularly SPR, can only handle a single kernel at a time. Because of this, the comparison in this chapter will focus on the final optimized Mosaic 1.0 architecture presented in [Van Essen10]. Despite this, some features required by the consensus architecture, such as a more flexible modulo counter, will be shared with Mosaic 2.0. The most relevant architectural details from the Mosaic 1.0 design can be seen in Figure 6 and Figure 15. The FU here is a "Universal FU" featuring a MADD unit in addition to standard ALU operations, which explains the four input ports.

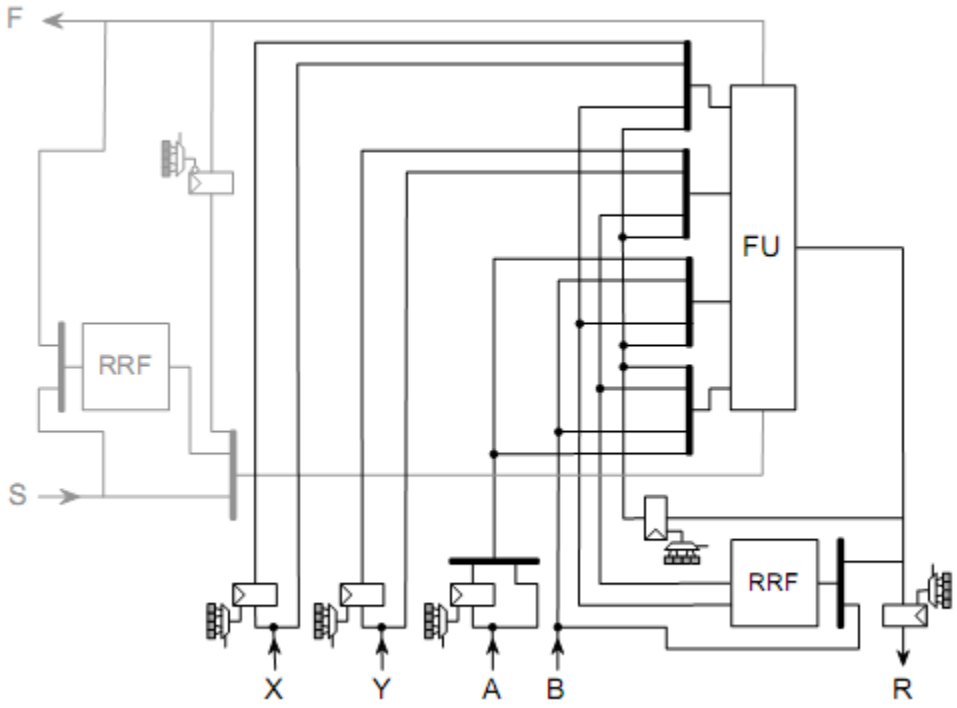


Figure 15. Optimized Mosaic 1.0 PE with Universal FU. Gray components are control path. [Van Essen10]

Table 2 compares the components required by the simple Trimaran processor to those available in Mosaic 1.0 and points out any significant challenges. At a very high level, there is a straightforward one-to-one mapping of Trimaran architectural components to Mosaic ones. For example, a functional unit performs similar operations in both cases, and both architectures uses rotating register files. However, as the table suggests, at a deeper level there are significant architectural challenges. Most of these involve connectivity in some way. The fixed connections between resources in Mosaic, unlike the highly configurable interconnect of an FPGA, are heavily optimized for low power high frequency CGRA operation. In many cases the connections required for the use of the various resources in a processor configuration are simply not present.

Trimaran Component	Mosaic Mapping	Challenges
Functional Unit	2 ALU+2 MADD for VLIW	Marshaling data
GP register file	Local rotating register file	Limited read ports
Branch target register file	Local rotating register file	Connectivity to PC
Predicate registers	Predicate register file	
Constant generation	Register slots	Constants burn registers
Memory	Cluster memory	
Control registers	Modulo counter, registers	Significant changes
Control logic	S-ALUs near PC	Data to control registers
Interconnect	Crossbar and PE muxes	Limited connection in PE

Table 2. Comparison of Trimaran and Mosaic components

Before addressing these issues in more detail it is important to understand the Mosaic architecture and the design decisions made more thoroughly. The resources available in the Mosaic cluster of Figure 6 break down into 32-bit word width resources (shown in black) and 1-bit predicate resources (shown in gray). The breakdown of these resources can be seen in Table 3. Note that all of the PEs feature 32-bit registers and 8-entry rotating register files as shown in Figure 15. Some of these registers have bypass muxes, but some do not for timing closure reasons. The PE also includes 1-bit input and output registers and a 1-bit 16-entry rotating register file.

32-bit data path (words)	1-bit control path (predicates)
Two 32-bit PEs with ALU and MADD	Two 1-bit PEs with 3-LUT
Two 32-bit PEs with S-ALU only	
32-bit 16-entry rotating register file	1-bit 16-entry rotating register file
Two 1K-word local memory	
32-bit wide crossbar	1-bit wide crossbar

Table 3. Word and single bit Mosaic cluster components

Each of these resources was chosen specifically to give Mosaic good performance at the lowest energy. For example, a rotating register file was chosen over distributed registers, a retiming chain or shift registers because of its high energy efficiency. Unfortunately, the limited read ports on this register file make it difficult to map some Trimaran instructions. Resources for storing short lived CGRA values were pushed close to the ALU, as can be seen in Figure 15. Although some of these registers help break up critical paths that would otherwise reduce the operating frequency of the CGRA, this was also done to reduce the total number of crossbar ports in a cluster. The crossbar is a high energy structure, and anything that reduces its utilization can have a significant effect on energy. For example, simply adding the local feedback register to the functional unit, without the local register file, reduces dynamic energy by 7% [Van Essen10].

Again, this sort of optimization is very important for efficient CGRA execution, but costly in terms of flexibility. For Trimaran mode, resources that are directly connected to the crossbar are much easier to map to their processor equivalents. Going through extra registers is difficult and the placement of some of these registers in Mosaic can lead to uneven delays between various operations, beyond what the Trimaran compiler currently supports. This conflict between CGRA energy usage and the ability to execute Trimaran operations is at the heart of the next section.

3.3 Consensus Architecture Design

The consensus architecture is one that combines the high performance and low power of the Mosaic 1.0 CGRA execution mode with the ability to execute Trimaran kernels like the Trimaran custom processor described in section 3.2. Because the performance critical components of most applications are expected to run in CGRA execution mode, the

consensus design approach attempts to preserve CGRA mode speed and energy whenever possible. In addition, Trimaran structures are kept simple and existing hardware or Mosaic components are used whenever possible. This should help to minimize additional design and verification time required for the Trimaran components. Ultimately, each aspect of this design is focused on reconciling the needs of the two execution models (discussed in 1.1.1), with preference being given to CGRA mode when required. With this in mind, the design of the hybrid modulo counter / PC is a logical starting point for looking at the consensus architecture.

3.3.1 PC and Modulo Counter

The basic architecture of a Mosaic 1.0 modulo counter can be seen in Figure 16. Under this design, the modulo counter counts down, using the subtract one hardware on the right, until reaching zero, at which point the comparator will configure the mux to select the II – 1 input instead. This design is simple for selecting the correct phase in Mosaic 1.0, and the modulo counter output can simply be routed to configuration memory to configure the various CGRA components. It can also be used in more complex designs, for example the predicate aware sharing described in [Friedman11]. As that work mentions, there will be many identical modulo counters distributed throughout the array. Aside from stalling, distributed modulo counters are not an issue because all of the modulo counters will be operating in lockstep.

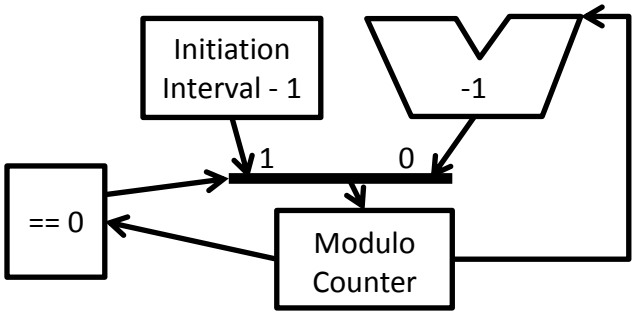


Figure 16. Mosaic 1.0 modulo counter

As discussed earlier, the Trimaran program would be stored in configuration memory in the hybrid architecture. Although this imposes significant restrictions on program length, given a maximum II on the order of 128, it is essential to maintaining as close to single-

cycle execution as possible. If the program was stored in a memory in the encoded operation form produced by a traditional linker, that would necessitate a decode stage. Storing the program in configuration memory means that the post-decode version of the instructions, as produced by the Verilog emitter, is already in hardware. Given this advantage, the limited storage of the configuration memory seems like an acceptable price to pay in the hybrid architecture. If this proved to be too limiting in the future, the maximum II could be increased or a decode stage and hardware could be considered.

With the program stored in configuration memory, one simple hybrid architecture approach would be to replace the Mosaic 1.0 modulo counter with a more complex version, capable of also accepting a branch input. This counter would effectively become a PC. This PC design can be seen in Figure 17. First, note that this version uses an up-counter, instead of a down-counter as in the Mosaic 1.0 modulo counter. This allows the same plus one logic to be used for the PC and the modulo counter. An additional mux selects the branch target, as read from a branch target register file, instead of the 0 value when in Trimaran mode. The mux which selects either the PC + 1 or the branch target value now detects II – 1 in CGRA mode or a branch instruction in Trimaran mode.

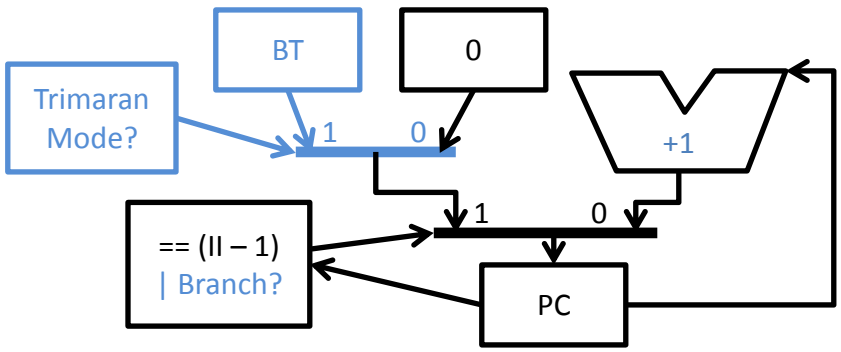


Figure 17. Proposed Trimaran PC (new components in blue)

Although this approach seems simple, it does require substantial additional hardware. First, the II – 1 comparator requires more logic than the Mosaic 1.0 modulo counter comparator which only had to test for zero. This approach also requires an additional mux. Most importantly however, the branch target and the predicate bits that specify if a branch should be taken on a given instruction must be routed to the PC logic. This

introduces additional load on the outputs of the branch target register file, which is a standard register file during CGRA execution. Finally, an extra input from the control network and some logic is required to select from between the PC + 1 (branch not taken or regular instruction) and the branch target (branch instruction and branch taken) for the new value of the PC.

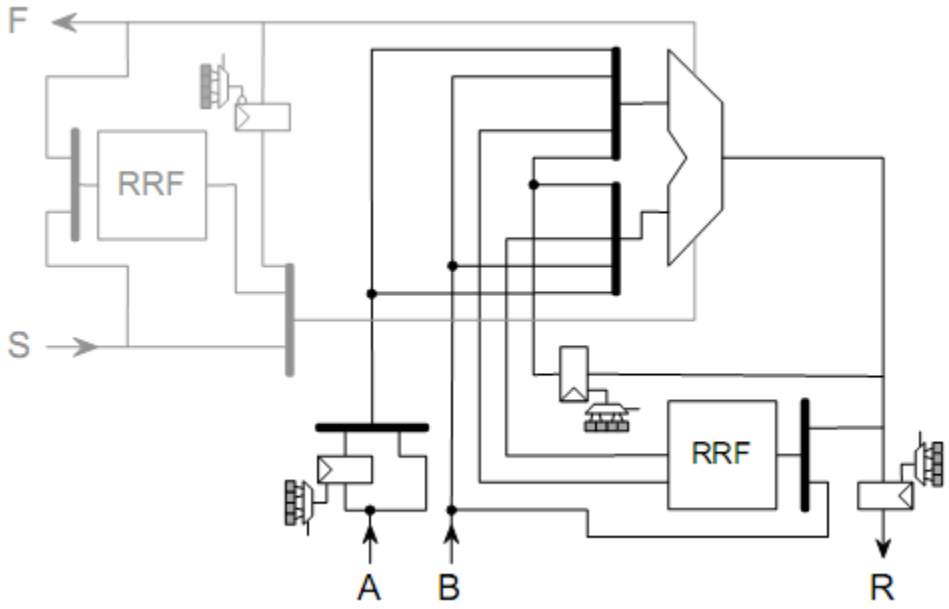


Figure 18. Optimized Mosaic 1.0 PE with S-ALU [Van Essen10]

As an alternate approach, one of the optimized PEs from Figure 18 could be used to augment the PC logic. The S-ALU in these PEs is similar to the one available in the Universal FU, but it lacks support for multiplication operations. In this case, the modulo counter could count down as normal in CGRA mode, and the S-ALU could be used to implement the PC increment logic in Trimaran mode, eliminating the need for dedicated hardware to compare to an arbitrary value (II – 1). The rotating register file local to this PE could be used as the branch target register file, and only a few additional muxes would be required on top of the Mosiac 1.0 modulo counter logic. This implementation is shown in Figure 19. Notice that the feedback register can be used as PC storage. It is also important to note that this implementation is only possible if the predicate aware sharing hardware presented in [Friedman11] is available. Otherwise, the S-ALU is incapable of performing a branch operation, since this would require selecting the PC + 1 or the

branch target without any hardware to generate PC + 1. With predicate aware sharing this could be done in the mux shown in navy on the figure.

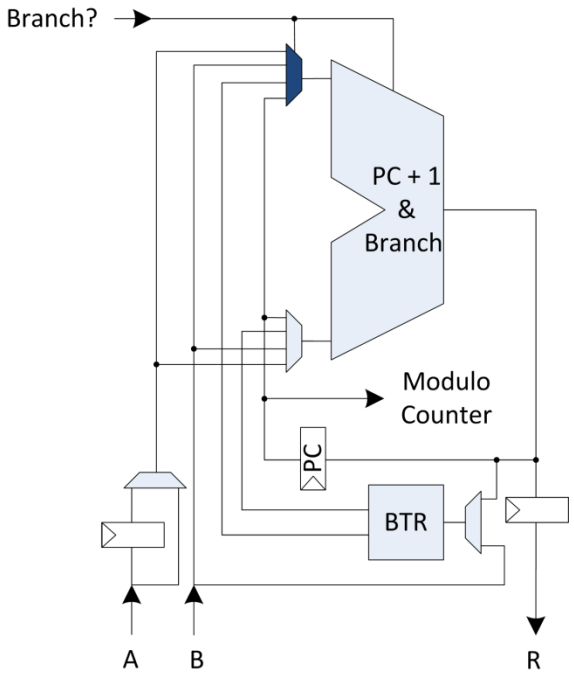


Figure 19. Mosaic 1.0 modulo counter combined with a PE to form a PC

Although this design would save on comparator logic and muxing, the fact that it is not possible on the optimized Mosaic 1.0 hardware that only allows predicate input to the S-ALU, and not the proceeding multiplexers, is a major strike against it given the design goals. Additionally, the hardware savings are not overly large. For these reasons, the most reasonable PC design to use is the first one described in this section. This design, where the modulo counter hardware is augmented directly to support Trimaran execution, also has a significant advantage in that it leaves another PE free for VLIW execution. Finally, some instructions, such as branch and link, require storing the PC back into the branch target register file. This would be difficult if the same S-ALU output wrote the branch target register file and the new PC. Many other decisions stem from the design of the PC, starting with the implementation of branch instructions.

3.3.2 Branches

Although the resource constraints discussed in 3.2 are significant, the most difficult Trimaran instructions to execute are some of the complex branches. The way that these

instructions can best be handled impacts the choice of resources used for other Trimaran components significantly. Two instructions are particularly troubling. These are the BRF and BRW instructions, designed to be used when software pipelining loops.

BRF operations are used with loops that run a known number of times, such as a typical `for` loop. This instruction decrements a special control register called the loop counter (LC), which specifies the remaining number of times to execute the loop. After LC iterations another control register (ESC) is used to specify the number of further iterations required to drain the pipeline. A predicate output is produced, which specifies if the execution has reached the epilog stage (LC = 0) or not. The BRF instruction also rotates the rotating register file.

The BRW instruction is just like the BRF instruction, except that it takes two predicates as inputs and doesn't use the LC. BRW is designed to be used in the case of loops that terminate on a condition (like while loops), and the first of these predicates specifies if the loop should continue (termination condition not met). The second predicate tells the loop to enter the ramp down phase. This is required because, unlike the LC which, upon reaching zero, will retain the zero value until the loop is complete, the first source predicate could potentially change after the loop has entered the ramp down phase. It's important to note that in reality if either of these predicates are false the loop will be in the ramp down phase. It's not important which is the loop condition and which is the ramp down marker. The operation of both loops is summarized in Table 4. Finally, both of these instructions have versions supporting any combination of branching or falling through on true conditions for each phase (loop, ramp down and stop).

Operation	Inputs	Outputs	Results
BRF	BTR	1 Pred	Rotates, Branches, Dec LC or ESC
BRW	BTR, 2 Preds	1 Pred	Rotates, Branches, Dec ESC

Table 4. Operation of BRF and BRW instructions

Of these two instructions BRF is the most difficult. BRW can be viewed as a subset of BRF in the sense that it doesn't use the LC and the AND of its two input predicates serve exactly the same role as LC > 0 in BRF. For this reason, any consensus architecture

designed to execute BRF operations can also execute BRWs, assuming the LUT on the control path can be used to combine the two predicates. There is also a branch on zero loop count (BRLC) instruction, but it is also a subset of the BRF instructions and can be implemented as such.

The BRF instruction (Figure 20) fundamentally requires a few operations. First, the LC must be compared to zero. Second, if the loop counter is zero, the ESC counter must be compared to zero. Third, the correct side effects must be performed depending on which of these first conditions was true, including setting the output predicate, decrementing one of the counters and rotating the register file. Finally, the branch must be taken or not as specified by the specific instruction. Each of these elements requires some hardware, but ideally it would all execute in a single cycle on a single PE.

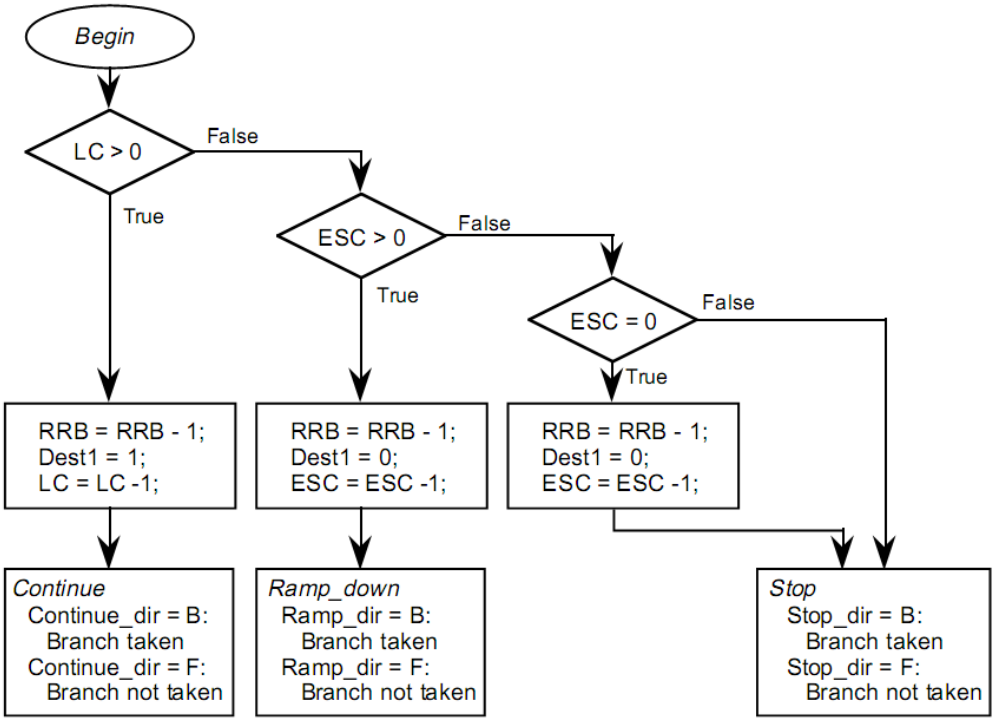


Figure 20. Diagram of BRF operation [Kathail00]

The resources used to do the two comparisons against zero depend on the storage used for the counters and vice versa. To do both comparisons at the same time, two comparators are required, but this is not unreasonable since the PE handling branches will be tightly coupled with the PC out of necessity. This means that the II – 1 comparator,

otherwise unused in Trimaran mode, can perform one of these operations if the LC or ESC is provided to it with a single additional mux. The other comparison will be performed by the S-ALU. In order to be close to these comparisons the ESC and LC counters can be stored in the A input register and feedback register respectively. It makes sense to put the LC in the feedback register since it is used by more instructions, and the path to the input register is both more expensive and slower due to having to traverse the crossbar. This delay in writing the escape register could be removed by adding a bypass mux to the PE output register (shown in red in Figure 21), but this might increase the length of the logic's critical path. In addition, it is probably not terribly significant since two BRF instructions in a row would not be particularly useful.

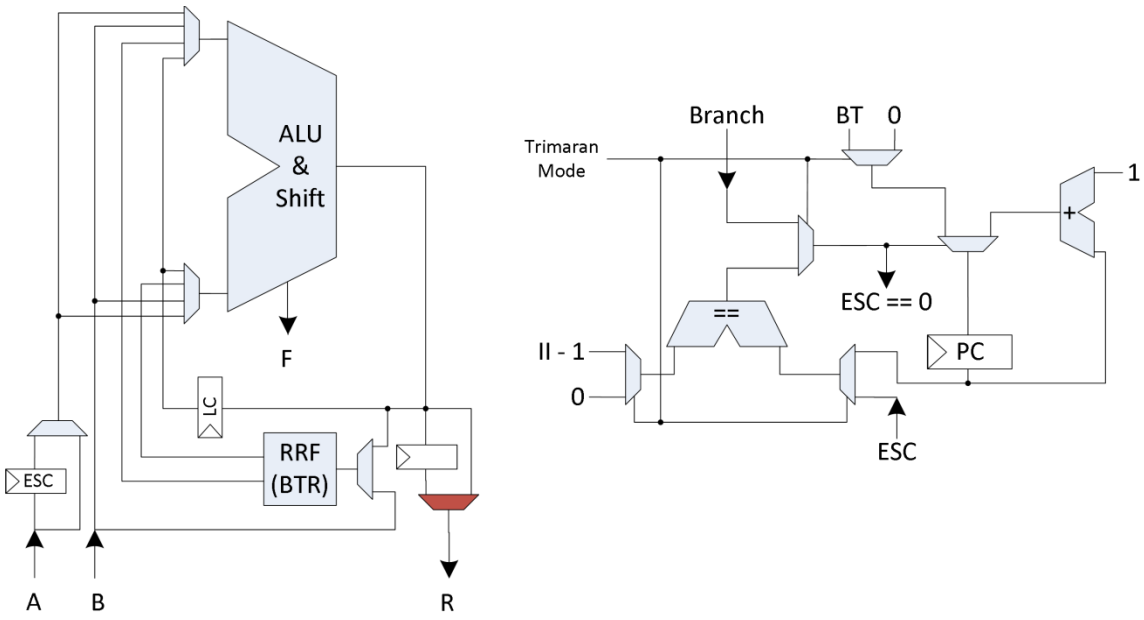


Figure 21. Branch operations mapped to a single PE and coupled to PC

Even with ESC and LC handled as described above, and shown in Figure 21, it would be nearly impossible to handle the entire loop in a single cycle. After performing both comparisons, the results have to be serialized (ESC = 0 will not end the loop if LC ≠ 0) and one of the counters must be decremented. The next PC must also be updated correctly. The first part of this is not difficult because, as seen in Figure 21, the control network output port from the S-ALU is not registered (port F). This allows the result of both comparisons to reach the 3-LUT in the control network through one additional mux and a connection to the output from the PC comparator logic. The LC or ESC can then be

decremented on the next cycle, as appropriate, although in the case of the ESC this introduces an additional cycle of delay to travel across the crossbar as discussed above. The output predicate bit can also be written to the predicate register file during this cycle.

The branch itself must also be taken or not taken correctly. To achieve this the PC must have either the PC + 1 or branch target address values muxed in correctly. On the second cycle, the predicate bit generated by the 3-LUT can be used for this purpose, although this does require the introduction of a potentially long delay path from the LUT output register to the PC. The relevant control bit is Branch in Figure 21 and S in Figure 18.

All other branch operations, ranging from the unconditional branch to the predicated branches, can be performed using subsets of the BRF logic. These simpler branches should also be performed in two cycles, even when not strictly necessary, to avoid introducing variable delay branches. Trimaran can schedule instructions in branch delay slots, but the MDES does not easily allow for branches of various lengths. One advantage of using two-cycle branches is that any register file can be used as the branch target registers with no issue. The only other branch that introduces other considerations is the branch and link (BRL). BRL must store the pre-branch PC back into the BTR file. This can easily be achieved by adding a connection from the PC register to the crossbar or the PE that stores the BTR file. This path will also be used for prepare to branch PC-relative instructions, among others. Legally, the PC is available as a control register which can be used as an operand for most classes of instructions. Although the PC may not be written as a general control register, LC and ESC can be. Given the architecture described in this section, writing the LC would occupy the ALU in the branch PE for one cycle. For this reason, the LC cannot be written in the cycle immediately preceding a branch instruction.

3.3.3 ALUs, General Purpose Register Files and Constant Generation

Although the branch target register file has some degree of freedom regarding its location, and the predicate register file and local memory also map trivially, the general purpose register file presents a significant challenge for multiple reasons. Some instructions, such as load and increment, have two register operands and produce two outputs. Some instructions, such as moving a GPR to a CR with a mask have two general

register or constant operands and a control register operand. These operations begin to run up against some of the basic port limitations of the optimized Mosaic 1.0 design.

The most straightforward solution to the GPR would be to place it in the large rotating register file on the crossbar. This approach has two major drawbacks. First, every register operation would go over the crossbar, burning significantly more power. Second, a simple string of instructions such as:

```
r3 <= add r1, r2

r4 <= or r1, r3
```

would run into significant issues. Even if only one PE was in use, given that the output of each PE is registered before returning to the crossbar, the updated value of r3 could not be stored in time to be used by the `or` operation unless the compiler was made aware of the feedback register. In the case of multiple PEs, this scenario would imply some sort of forwarding network before the crossbar, which would be both complicated and expensive. Alternatively, a bypass could be introduced on the output register of the PE, as discussed in 3.3.2, but this introduces timing issues. A dedicated output from the two PEs featuring MADD units to the large register file could allow for writing without a bypass, but it would introduce some loading of its own in addition to an extra mux.

Another option is to use the rotating register file inside the PE as the Trimaran's GPR. This approach gets around the issues mentioned above since there is a single cycle path from the rotating register files, through the MADD (when used for single-cycle operations) and back to the register file. However, this would create significant VLIW issues. It would be challenging for the values in one PE to get to another in a timely fashion, as at least one extra cycle would be required to get values from the output of one PE to the inputs of another. Register file to register file transfers would be even worse, requiring another additional cycle. Unless large amounts of parallel work on independent data was available, this would significantly reduce the benefits of having a VLIW execution mode. Some of the options discussed above, including bypassing output

registers, would be an option here, but if those solutions were being considered, the larger register file seems like a more natural fit.

There is also an issue regarding constant values. In Mosaic 1.0, constants are preloaded into local register files before execution begins, and read from there. SPR could potentially schedule multiple runtime constants into the same phase, so the unoptimized PEs featuring U-FUs have four read ports on the register files [Van Essen10]. The optimized version of the U-FU PE drops two of these read ports, suggesting that SPR should limit the maximum number of constant inputs to the same U-FU in a cycle to two. This constant mechanism is awkward for Trimaran. If it was used, register files would lose capacity for every constant operand used in the program. Not only is this a feature the Trimaran compiler doesn't support, but even if it did, register file capacity is a much bigger issue for a Trimaran program than an SPR one. On the other hand, constant generators would require large amounts of configuration memory. At 32-bits wide, and given that two constant generators would be required, this would introduce 1 Kbyte of additional SRAM in addition to extra muxes and connections to the configuration control network.

It is clear from this discussion that for any consensus solution with a functional VLIW mode, even just 2-wide, the changes required to the GPR and constant generation will be among the most costly. Despite this, adding bypass registers to the PE output appears to be the best solution. None of the other proposed solution would allow for single cycle VLIW operation, without adding significant additional scheduling burdens to Trimaran. Instead, adding output register bypass will allow the large register file attached to the crossbar to function as the GPR, and values can easily be shared among FUs operating in VLIW mode. This solution also has the advantage of providing a single location where more ports could be added as necessary to support wider VLIW, instead of having to potentially add ports to register files in each PE.

One additional property of this solution is that it leaves the local rotating register files in each PE unused. This could provide a good location to store constants, although it would place an arbitrary constant limit on programs. The fact that the ALUs can generate

constant values of 0 and 1, along with the limited number of instruction slots available to a Trimaran mode program make it unlikely that the local register file would be filled with constants. If it was, the compiler could be modified to store additional constants in some of the many other registers available in the fabric and generate any additional constants required through runtime arithmetic. These solutions should eliminate the need to add constant generators to the crossbar or the global register file. Adding ports to the crossbar itself is costly, as is the configuration memory requirement of constant generators, so any solution that does not require them is extremely beneficial.

Mosaic 1.0 predication implements mutually exclusive operations, both of which are performed, and the predicate bits are used to select the correct output (see 1.1.1). Most Trimaran operations can be predicated, but the semantics are different. A Trimaran operation that has a sense that doesn't match the predicate (for example the predicate bit is false, and the operation is predicated to execute on true) is simply treated as a nop: it should have no effects. This form of predication is also called guarded execution. The method for implementing this in the case of branches is discussed in the previous section, but for other operations such as arithmetic Mosaic 1.0 hardware is insufficient. Instead, a predicate enable will have to be added to state-holding elements such as the register files. This could use existing logic in the case that predicate aware scheduling hardware is present, but in its absence new hardware would be required. Given that the predicate bits are already available on the control crossbar, the worst case scenario would be an extra gate prior to the enable and an extra crossbar read port. Note that structures other than the register files, such as the local memory, might require enables in this case.

A summary of the hardware changes proposed in this section can be found in Figure 22.

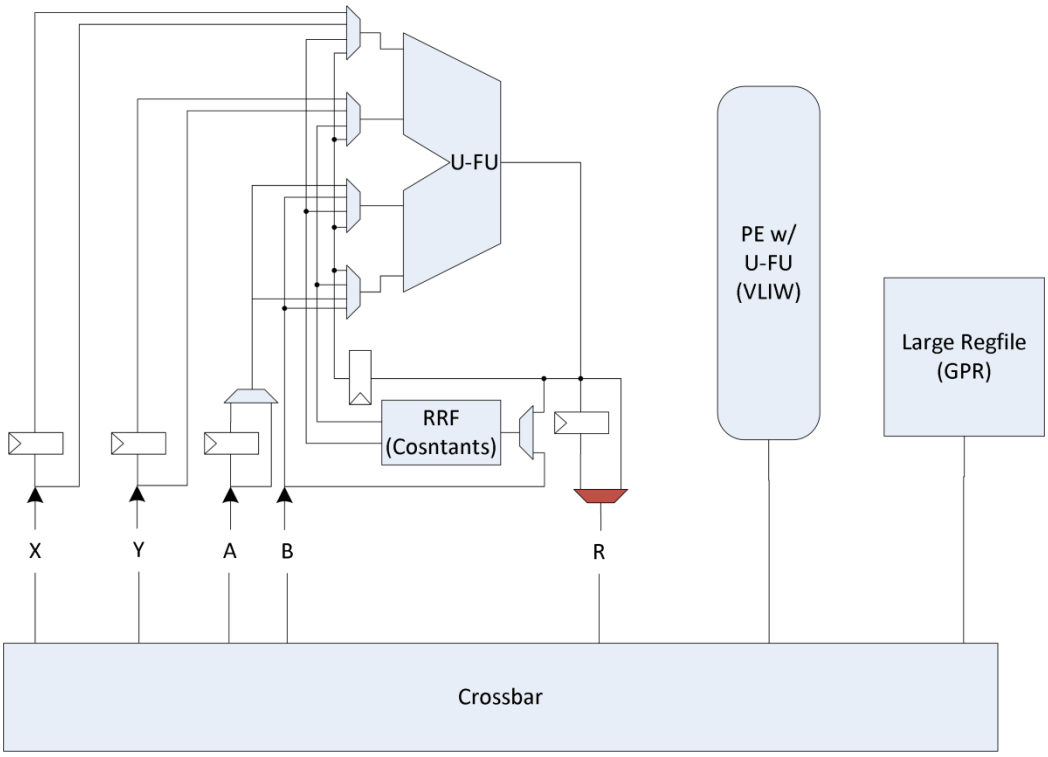


Figure 22. Hybrid PE with U-FU and connecting register file for VLIW execution

3.4 Consensus Architecture Analysis

Section 3.3 describes the design of each hybrid component in detail. The result of all of these changes to the optimized Mosaic 1.0 design is the "hybrid" architecture. This architecture can be divided into four major components: execution support (modulo counter / PC), branch support, predicate network and data network. The design of each of these components will have a significant impact on the performance of one or more classes of Trimaran instruction.

The changes to the modulo counter are the most extensive, since its purpose is shifted dramatically from a simple counter to a full-fledged PC surrounded by additional branch logic. However, the amount of additional hardware is not excessive. All that must be added is four muxes and a few additional connections from the branch PE and the predicate network. The most expensive new component is the comparator, which must be modified to compare against an input value instead of 0.

The data network also avoids significant modification, although adding a crossbar port to support more register file read or write ports should they be required, would be costly. The PEs themselves require only very minor modification; now both U-FU and S-ALU PEs must have a mux on the output path. Additionally, in the case of Mosaic hardware without predicate aware sharing, a connection from the predicate network to the write enables of various storage elements would be required to implement Trimaran style predication.

In general, the changes described above meet the goals set out for the consensus architecture. Specifically, the modifications should not have significant impact on the area or operating frequency of the Mosaic hardware. Most of the additions were a few muxes and extra communication channels. Although these components are not insignificant, especially considering the extra configuration memory required, the total percentage impact should be small. Frequency will likely be slightly reduced due to the additional load of this extra logic, but since no paths were introduced that require multiple operations or traversals of the crossbar in a single cycle, both CGRA and Trimaran modes should continue to operate at a reasonable clock frequency.

Most importantly, the energy and performance of kernels executing in CGRA mode should not be significantly affected by any of the changes introduced to allow execution of Trimaran kernels. All of the additional logic discussed can be configured to a fixed state during CGRA execution, essentially eliminating dynamic power consumption. Other techniques, such as power gating the constant generators or other Trimaran only components, can even reduce the leakage of these components and help CGRA mode in the consensus architecture operate very close to Mosaic 1.0 levels.

One area where the results are less positive is the consensus architecture’s execution of Trimaran kernels. Although most Trimaran instructions will remain single-cycle, this was not possible in all cases. Branches will require two cycles as discussed in 3.3.2, and Trimaran can fill the branch delay slot with other instructions. For most ALU operations single-cycle execution will be possible, as seen in 3.3.3, but the Mosaic MADD unit requires two-cycle multiplication. Since Trimaran will be relying on the MADD located

inside the Universal FU for all multiplication operations, the speed of these operations will be limited to the execution speed of the MADD. Although it would be possible for Trimaran to take advantage of pipelined multiply instructions, given the Mosaic 1.0 multiplication hardware, the actual amount of performance that can be regained in this manner depends on the scheduling ability of the Trimaran compiler and the inherent pipelineability of the program being compiled.

Memory access also presents some minor challenges to the single-cycle assumption. Trimaran only supports memory operations to addresses already stored in a GPR, so support for address calculations prior to reading from the local memory was not required. However, there are post-increment memory operations in Trimaran. These instructions load or store to an address read from a register, and also compute a new address that is at an offset from the original one. This computed address is stored back into the register file along with the result of the memory operation, in the case of a load. The register file write port pressure of instructions with two destination registers has already been discussed. Since the computation of the new address can occur in an ALU that is otherwise unused during a memory load or store operation, there is no reason that both of these things can't occur in the same single-cycle. Because of this, the single-cycle assumption is intact for memory operations, even post-increment load and store.

Finally, VLIW execution could present issues for the majority single-cycle model. Although there is currently an unused S-ALU in the consensus design, VLIW beyond 2-wide could be challenging within a single cluster due to resource constraints, the most significant being register file ports. Any attempt to add inter-cluster communication to the execution of a single Trimaran kernel in VLIW mode would incur many cycle delays. The cost of traversing the inter-cluster interconnect would be great, and tremendous compiler modifications would be required to add scheduling support for this scenario to Trimaran. It is difficult to imagine that the performance gain of 3 or 4-wide VLIW execution could overcome the penalty of the added delay.

4. Modeling and Methodology

A working system, at the level of Verilog simulation, was constructed only for the custom Trimaran processor. This required that the performance of the consensus architecture be estimated based on the analysis in 3.4.

4.1 Testing Methodology

Performance testing was straightforward on the single-cycle Trimaran processor. First, the HPL-PD machine description file was set to match the architecture of the custom Trimaran processor. Next, the desired kernel was tagged with marker functions and run through the scripts described in 2.1. These scripts produced Trimaran-ready C code, which was then compiled using Trimaran, producing both Verilog output and all of the support files required to run Simu. This entire flow can be seen in the Trimaran path of Figure 12.

Verilog simulation, being very slow, was only used to verify correctness. Performance for longer simulation runs of the Trimaran only architecture was measured using Simu. Because Simu is fully aware of the architecture, as specified in the HPL-PD description, and there are no uncertainties such as those introduced by a complex memory hierarchy, its simulations are cycle accurate just like the Verilog runs. Also note that read streams were treated as always having data, and write streams as always having space. In other words, no stalls were permitted to occur during these simulations.

Testing SPR performance was similarly straightforward. Using the current Macah to SPR tool flow, as shown in Figure 7, the benchmark application was run through Macah. Next the desired kernel was selected and run through the fsim Macah simulator. This simulation provides an iteration count for the design. Next, the SPRanalyze tool runs SPR with no resource constraints. This determines the minimum recurrence II of the kernel. This is the lowest possible II that the kernel can have in its current form, due to dependencies in its dataflow graph. Note that the minimum II found by SPRanalyze is not necessarily something fundamental to the application. Instead it is simply based on the Macah output from the current implementation. Improvements to the code can frequently be used to improve this II lower than its current point. Lastly, SPR itself is run against the

kernel, given an architecture very similar to Mosaic 1.0 as described in this thesis and in [Van Essen10]. SPR will determine the actual II of the kernel for some number of clusters, based on the resources available. SPR uses heuristic algorithms, so the II found in this stage is not an absolute minimum for the given kernel on the given hardware. Instead, it is a best effort attempt by SPR in a reasonable runtime.

Once all of this data is collected from the simulated execution of the kernel, relative performance can be gauged. For the single-cycle Trimaran processor model, cycles of simulated execution time are equivalent to clock cycles in the final design. Things are a little more complicated for the SPR kernels. Ignoring stalls, since we've already established there won't be any, the cycles of execution for those kernels running in CGRA mode is essentially the iterations multiplied by the II. This makes sense because each iteration requires II cycles to complete, so:

```
II * iterations = cycles / iteration * iterations = cycles
```

In reality, even without stalls this calculation is off by a small amount. This is due to the time required to fill the pipeline at the beginning of execution and drain the pipeline at the end of execution. However, given a sufficiently large number of iterations, this small discrepancy can safely be ignored. The final result of these calculations is a simple comparison between the execution cycles of Mosaic 1.0 CGRA mode and the single-cycle Trimaran custom processor.

4.2 Consensus Architecture Performance Estimation

The execution cycle comparison method should also provide the performance of SPR on the consensus architecture. Great care went into the design to preserve CGRA performance, and from the design analysis it appears that it does not, in fact, reduce performance in any significant way. Because of this, the cycle counts from the previous section should still be applicable.

Unfortunately, determining the performance of Trimaran execution on the consensus architecture is not nearly as simple. The consensus implementation will not be able to execute all instructions in a single cycle as the Trimaran custom processor did. Instead, it

has some multi-cycle instructions, some of which can be pipelined easily and some of which cannot. To complicate things further, some of the restrictions introduced by this pipelining would require substantial changes to the compilation and optimization phases of Trimaran to implement. For this reason, consensus architecture Trimaran performance is estimated, as described below, using the worst case assumption of no pipelining. The Trimaran consensus results must then be considered as a lower bound, where the performance of an actual implementation could be somewhat closer to that of the single-cycle Trimaran custom processor.

The branch class of instructions is an example of these challenges. Section 3.3.2 states that all branches are two cycle instructions and, as in many architectures, two branches can't be issued in adjacent slots. However, since unrelated functional units can operate in parallel in Trimaran, instead of a branch delay slot other instructions that don't affect the branch values could be issued in both slots. To make things even more complicated, for branches that use LC and ESC, these values must be written the cycle before the branch begins. However, since there is no path to the feedback register in the PE (used to store LC, Figure 21) that doesn't first travel through the S-ALU, this value must be written some time before the branch begins, without overlapping with any other branch instructions or writes to the other control registers. It is additional constraints like these that make developing an efficient schedule for Trimaran on the consensus architecture more difficult than simple HPL-PD description modifications. The execution of each instruction type is summarized in Table 5. Note that this thesis frequently groups the details of the HPL-PD instruction set that Trimaran implements under the term "Trimaran", although it can actually support many other ISAs as well. Details of the HPL-PD instructions can be found in [Kathail00].

Besides instruction classes that have inherent difficulties requiring design changes, there are some complex instructions that require two cycles to execute. For example, Trimaran features fixed shift and add instructions as well as logical operations that compliment one of the inputs. Without ALU modifications, these instructions can be trivially mapped to a single PE, but they are not pipelineable. This could present a scenario where all integer instructions must be modeled as two cycle operations for the consensus architecture, but

since these instructions are not observed in any of the benchmarks used this situation was avoided. Instead of creating various execution lengths for different integer operations, it may be better to remove these instructions entirely. The only cost to removing these instructions and forcing the compiler to output two different instructions is code size (important given Mosaic's 128 configuration slots), register file pressure and possibly crossbar energy. The alternative is potentially costly ALU modifications that SPR doesn't take advantage of. Removal is then a particularly good solution if these instructions are used infrequently, as appears to be the case.

Class	Cycles	Pipelineable	Notes
Integer	1, 2	No	Some instructions require two ALU ops
Multiplication	2	Yes	
MADD	2	Yes	Not currently implemented in Trimaran
Floating Point	-	-	Not supported on Mosaic hardware
Conversion	1	Yes	
Move	1	Yes	All moves must go through a PE
Compare	1	Yes	
Memory	1	Yes	No speculative or reference hierarchy
Pre-branch	1	Yes	
Branch	2	Yes	See 3.3.2

Table 5. Execution cycles of Trimaran instruction classes on consensus architecture

5. Benchmarks

Three benchmarks with very different properties help to expose the relative performance of a kernel executed using Trimaran instead of CGRA mode. The code for these benchmarks can be found in Appendix A.

5.1 2D Convolution

In the case of a very simple 2D Convolution implementation in Macah, the convolution operation is performed on an input stream representing a 2D matrix. The same convolution mask is applied to each area of the input Matrix, to produce the output matrix as shown in Figure 23. The mask values are multiplied against each cell in the input matrix, and the cells surrounding it. Although a 3x3 mask is used in the example, a larger mask is also possible. The results of the multiplication are then summed, producing the final output value for the center cell. Note that there are additional complexities, such as handling cells on the edge of the input matrix and scaling the output by a constant factor, but these are not part of the kernel being benchmarked here, since they will be implemented in other kernels.

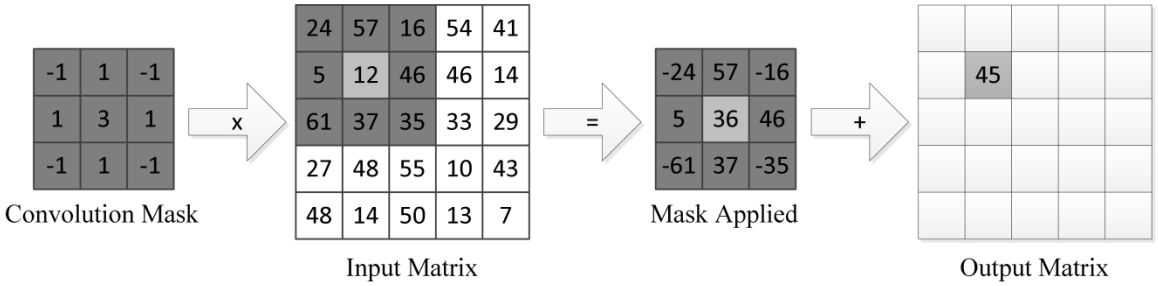


Figure 23. 2D convolution example

The convolution kernel considered here, `conv`, performs the multiplication and addition stages described above, using two `for` loops. This is a very simple kernel without much complex control flow. In addition, when written correctly there are few dependencies from one output value to the next, so the II should be low and the CGRA performance should be high. On this kernel, Trimaran results are expected to be much worse.

5.2 Bayer Filter

The Bayer filter is a critical part of the typical digital camera pipeline [Bayer76]. Images will often be captured by separate arrays of red, green and blue sensors (or shared sensors time multiplexed between colors). The purpose of the filter is to combine these separate color pixels into a single full color image. Depending on the exact makeup of the sensors, many variations on the filter exist, but the fundamental operation is always to combine the various sensor values into a single image.

When implementing the Bayer filter for a stream based CGRA like Mosaic, the two dimensional input matrix must be streamed into the kernel as a one dimensional input. This presents a problem along the edges of the input sensor data similar to the 2D convolution, and part of the solution to this problem is a kernel that mirrors the pixels along the edges, called `fillHorz`. In addition to the two `for` loops, this kernel contains an `if-else` block made up of five different conditions. This complex control logic can be expected to show significant benefit under Trimaran, since the Macah version of the kernel will be executing significant amounts of predicated logic in parallel before throwing most of those results away.

5.3 Discrete Wavelet Transform

In a discrete wavelet transform (DWT) an input matrix has a set of high and low-pass filters applied to it recursively [Daubechies90]. This operation is useful in many applications, including imagine compression algorithms such as JPEG. In the case of the DWT used in JPEG, the two dimensional input matrix has a high and low pass filter applied to it in the horizontal direction, followed by both filters being applied in the vertical direction. The result of this first set of transforms can be seen on the left hand side of Figure 24. The bottom right section has the high portion from both the horizontal and vertical filters (essentially a diagonal). The top right and bottom left section of the image now contains the horizontal high pass and the vertical high pass respectively. These high pass filters highlight the noise, or sharp changes, in the image. Similarly, the top left has both low pass filters. After scaling, this essentially creates a smaller version of the original image in the top left. Finally, the same sequencing of filtering is applied repeatedly to the top left. The first application produces the results on the right hand side

of the figure, and each subsequent application operates on an image a quarter the size of the previous one. For more details about the operation of the DWT, see [Fry01].

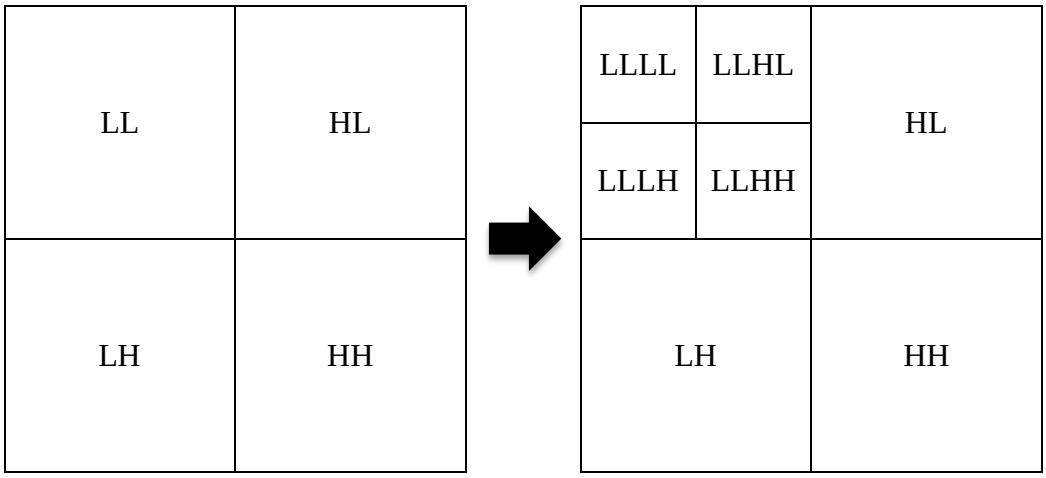


Figure 24. The first two stages of the 2D DWT used in JPEG image compression

One important aspect of the streaming Macah implementation of the DWT is that the image must be divided into strips on which to perform the filter, but the horizontal and vertical passes require these strips in different directions. Because of this, between the application of the filters, the matrix must be mirrored. The kernel we use as a benchmark, `leftVerMirror`, performs one of these reflection operations prior to the low-pass filter. To do this, the kernel must read in an entire strip of data and store it, write out the mirrored version of this data, and then write out the un-mirrored values to complete the strip. This requires a series of `for` loops, including nested loops for the mirrored data. This kernel has interesting control flow because each of these loops makes use of the same set of streams.

A final note about these benchmark applications is that they all appear to be similar at some level. Each of the three performs some operations on a stream representing a two dimensional matrix to be filtered. Despite these similarities, these are actually very different benchmarks because each kernel comes from a different part of the process and, most importantly, has a separate variety of control flow.

6. Results

This section presents the results of the benchmarks discussed in section 0 on the architectures from section 0 followed by a comparison of the architectures.

6.1 Optimized Mosaic 1.0 Performance

The first area to examine is the performance of the Macah / SPR flow on the optimized Mosaic 1.0 architecture. The architecture used to run these benchmarks is similar, but not identical to that described in [Van Essen10]. The discrepancy is due to the rotating register files and some of the distributed registers in the PEs being replaced with retiming chains. Although no performance numbers are provided directly, the energy difference between these approaches is between 5 and 10 percent. These results are close enough, that as long as the reader bears them in mind, the overall conclusions should not be skewed by the lack of a rotating register file.

These results are based on running each benchmark with multiple seeds on Mosaic 1.0 clusters ranging in size from 1 to 16. These clusters use only core tiles, not edge tiles. In the case where different seeds produce different results, only the best result is used. As described in 4.1, C-based Macah simulations are first used to determine the iteration count for each benchmark followed by determining the final II using SPR. These initial results can be seen in Table 6, assuming no particular cluster constraints.

Benchmark	Input Size	Iterations	Minimum Recurrence II	Minimum Resource II
Convolution	128 x 128	17,689	3	4
Bayer Horizontal	128 x 128	17,161	7	5
DWT Left Vertical Mirror	256 entries, 64 per stripe	35,029	2	1

Table 6. Optimized Mosaic 1.0 cluster count independent performance results

These results alone, without resource constrained IIs, don't provide very much useful information. It is worth noting that the iterations are very similar to the expected results (128 x 128 = 16384) and all of the minimum IIs are reasonably small. The true minimum II is the larger of the recurrence and resource minimums. The recurrence minimum II, as

discussed, is essentially the length of the longest dependence loop in the program's DAG. The minimum resource II is determined by shared resources that must be time multiplexed, in situations where more resources could not improve performance due to communication time or limited routing resources.

Figure 25 shows the II vs. cluster count for each benchmark. Note that as expected, the II decrease as the number of available clusters increases, approaching a minimum at around 8 clusters for most of the benchmarks. For more than 8 clusters, the II actually increase, perhaps as routing the more spread out placement becomes difficult. For more clusters, many more runs are required to overcome random noise as well. It's worth noting that the II is consistently somewhat high for 16 clusters (except in the DWT case) across many SPR runs with different seeds, so this probably is not just a case of randomly selecting a worse mapping of the program to the resources leading to a higher II. In the case of the convolution benchmark, the `FOR` loops imply a large amount of parallel computation. The inability to sufficiently serialize this arithmetic and memory access may be responsible for the failure to route on 1 or 2 clusters. The Bayer filter kernel very quickly approaches its minimum recurrence II when running on 2 or more clusters. This is not surprising because although it has a large number of conditional statements, all of them are very small and some perform similar work. Finally, the DWT mirror kernel follows the expected pattern of reaching an II close to its minimum at 4 clusters. It is worth noting here that there is likely another constraint, since the minimum achieved II of 5 is more than double the theoretical minimum of 2.

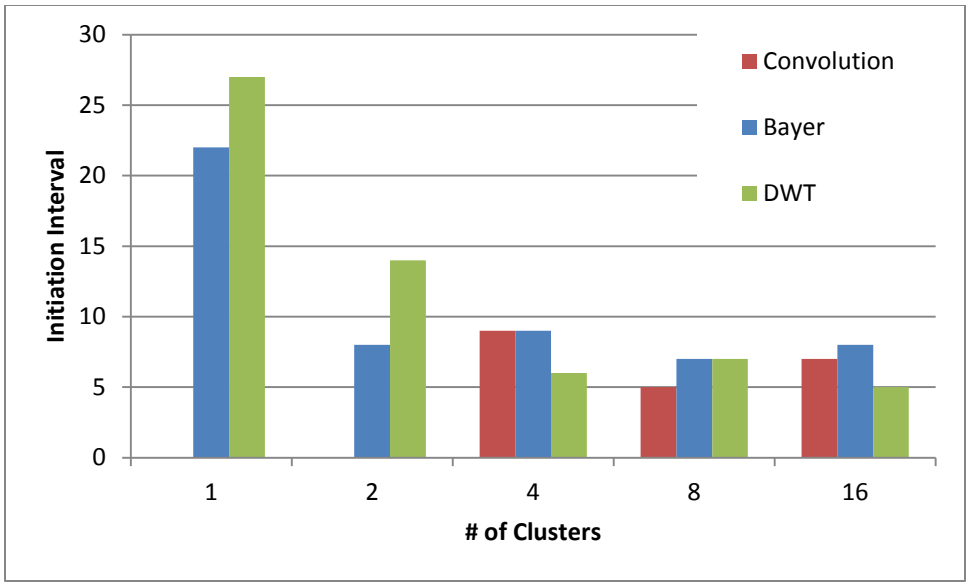


Figure 25. II vs. clusters. Convolution was not routable on less than 4 clusters

6.2 Custom Trimaran Processor Performance

The single-cycle custom Trimaran processor is much more of a proof of concept for Verilog code emission than an architectural benchmark. In fact, since it is specifically designed to perform all Trimaran instructions in one cycle, performance is identical to the Trimaran baseline for any given benchmark. The results of the benchmarks in Trimaran, after running the Macah to Trimaran C script, are provided in Table 7.

Benchmark	Input Size	Cycles	Optimized
Convolution	128 x 128	6,238,595	
Bayer Horizontal	128 x 128	402,240	67,220
DWT Left Vertical Mirror	256 entries, 64 per stripe	143,693	

Table 7. Single-cycle custom Trimaran processor benchmark results

Although these results are most interesting when compared to Mosaic below, there are some points that stand out. The Convolution kernel is clearly extremely expensive in Trimaran, which is not surprising since this processor is single issue and the kernel involves a large amount of arithmetic that can easily be parallelized. The results for the DWT kernel are much better, as is expected. Although there are many loops in this

kernel, they are not nested, producing a very serial kernel. There is also very little computation inside each loop. The Bayer kernel performs very badly in Trimaran.

6.2.1 Bayer Kernel Optimization

Given that the Bayer kernel is made up of a single large conditional it should perform reasonably well as a C program, but it does not do so. This is because the conversion from highly optimized Macah code to Macah simulation C and finally to Trimaran C does not always result in the most efficient Trimaran implementation of a program. This leads to the valuable insight that when implementing the same kernel in Trimaran and Macah, it may be worth writing each version separately in the manner best suited to the execution model.

The optimized Bayer code is available in Appendix A. In the Macah version, the two outer loops iterate over all columns for each row. Inside the column calculation, special conditional logic handles the first three and final three columns. The essence of the optimization is to remove these conditionals, and simply perform the special operations from inside the row loop prior to entering the column loop. The serial nature of this implementation makes it very well suited to Trimaran. Removing the many conditional checks of the large `if-else` statement greatly improves Trimaran performance, since now only the two `for` loops require branches. This sort of optimization could be performed by a more advanced compiler that fully analyses the code inside the loop. All optimizations that Trimaran supports are performed on the converted Macah code, but this particular optimization appears to be beyond the scope of the current compiler.

For the Bayer kernel, the functionally identical version written specifically for Trimaran performs 6x better (the optimized column in Table 7). Furthermore, given that automatic conversion of Macah kernels to Trimaran ones does not always produce good results, it may be that a designer should implement most kernels in the language best suited to the application initially and only use the automatic conversion scripts sparingly. Obviously, this advice does not apply in cases where the performance of the Trimaran kernel is completely irrelevant.

6.3 Consensus Architecture Performance

Although there is no implementation to benchmark, the results from the single-cycle Trimaran processor and Mosaic 1.0 can be extrapolated to the consensus architecture. In the case of CGRA execution mode performance, this is very easy. As discussed in 3.4, the CGRA performance of the consensus architecture should be identical to the optimized Mosaic 1.0, with the exception of power, not analyzed here. This only leaves Trimaran mode.

The consensus architecture executes most Trimaran instructions in a single cycle. The exceptions modeled here are branch instructions, which require two cycles, and multiply instructions, which occupy a U-FU for two cycles. For more information see 3.4. When accounting for this extra execution time, the results in Table 7 change slightly to those seen in Table 8. The impact of these changes will be examined in more detail in the following section.

Benchmark	Input Size	Branch	Multiply	Total
Convolution	128 x 128	540,282	435,600	7,214,477
Bayer Horizontal	128 x 128	117,280	0	519,520
DWT Left Vertical Mirror	256 entries, 64 per stripe	34,988	0	178,681

Table 8. Consensus architecture Trimaran performance

6.4 Analysis of Results

When looking at the consensus architecture performance there are two important aspects to examine. First, it is possible that some kernels will perform better overall in Trimaran mode. These kernels have a minimum II (either resource or recurrence) such that they will always be slower in CGRA mode, no matter how many clusters they are executed on. The second class of kernels may be able to perform faster on the consensus architecture when run as CGRA kernels, but this may require many clusters. In this case, analysis of performance criticality, free clusters on the array and energy usage would be required to determine which execution mode is best for that kernel.

For this second class of applications, it is useful to look at performance per cluster as a way to compare Trimaran and Macah / SPR execution resources. This measure is simply the execution cycles of the kernel multiplied by the number clusters it is executed on. Another way to think of this is as the total number of issues slots across all clusters used during execution. This metric can be considered for any number of clusters, but it will be most useful to use the best case for Mosaic here. Finally, it's important to observe that even if Trimaran is best for this metric that does not always mean a kernel should be executed in Trimaran mode. There could be cases where a kernel is the critical path in a multi-kernel application, in which case it should be executed as fast as possible regardless of the number of clusters required. Similarly, for non-performance critical kernels Trimaran execution on a single kernel may be desirable even if it is much slower in order to save energy and array resources.

Before examining CGRA execution mode, it is worth looking at the performance penalty of going from single-cycle instructions to some multi-cycle instructions. This data is available in Table 9. Of these benchmarks, only the convolution features multiplication. Surprisingly, despite the multiplication, it loses the least performance of all the kernels, with the consensus architecture performing at 86.5% percent the speed of the custom processor. This is due to the fact that the convolution has far fewer branch instructions, about 9% instead of 25% or 30%, when compared to the other kernels. For the Bayer and DWT kernels, the performance is reasonable close, ranging from 77.4 to 80.5%. Some of this performance loss could be mitigated by VLIW execution.

Benchmark	Single-cycle	Multi-cycle	Performance
Convolution	6,238,595	7,214,477	86.5%
Bayer Horizontal	402,240	519,520	77.4%
Optimized Bayer	67,220	83,474	80.5%
DWT Left Vertical Mirror	143,693	178,681	80.4%

Table 9. Trimaran performance loss in going from single-cycle execution to consensus architecture

Before these Trimaran results can be compared to CGRA mode results, the total number of issues slots used across all clusters must be compared as described above. Figure 26 shows the number of issue slots used for each case and these results multiplied across all clusters in use can be seen in Figure 27. Note that for every kernel the total time to completion is lowest for 8 or 16 clusters. This is not surprising as all of these tasks have sufficient parallelism available that performance can improve with more resources, and only the Bayer filter task ever reaches its minimum II. However, the last 4 or 8 clusters added only produce increment gains as the second chart shows. The total number of issue slots used across all clusters increases significantly after 2 or 4 clusters. This suggests that if the CGRA can be filled, only dedicating 2 to 8 clusters to any of these kernels might result in the best total performance per watt.

Figure 26. Execution cycles until kernel is completed vs. number of clusters used in CGRA mode

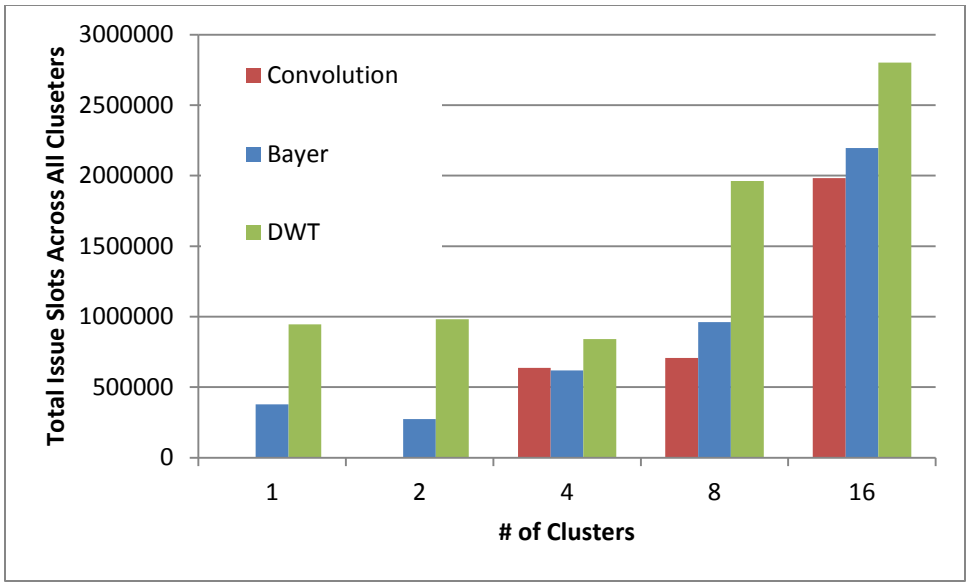


Figure 27. Total issue slots summed across all clusters executing a given CGRA mode kernel

Table 10 compares Trimaran performance on the consensus architecture to the best CGRA performance from Figure 26 for each benchmark. In the case of the convolution, the CGRA implementation is about 82 times faster. This is not surprising given the complete lack of control flow and large quantity of parallelizable ALU operation in the benchmark. In other words, this is an ideal case for CGRA execution. The Bayer filter is a much more interesting case. On 8 clusters, CGRA execution is 4.3 times faster for the direct Macah to Trimaran Bayer horizontal mirror kernel. However, the hand written Trimaran C kernel is actually faster than the CGRA version, executing in about $^{7}/_{10}$ of the time. This highlights the significant issues encountered when running a heavily optimized Macah kernel directly in Trimaran. However, this hand tuning will only work for kernels with significant amounts of control and serial work. No hand tuning of the C code is going to improve the performance of the convolution kernel by a factor of two in Trimaran, let alone make it better than the SPR results. Finally, the DWT kernel is only about 2% faster in on the CGRA, but it requires 16 clusters to reach this level of near-parity.

Benchmark	Trimaran	Clusters	CGRA Cycles	Runtime
Convolution	7,214,477	8	88,445	81.6 x
		4	159,201	45.3 x
Bayer Horizontal	519,520	8	120,127	4.32 x
		1	377,542	1.38 x
Optimized Bayer	83,474	8	120,127	0.695 x
		1	377,542	0.221 x
DWT Left Vertical Mirror	178,681	16	175,145	1.02 x
		1	945,783	0.189 x

Table 10. Absolute Trimaran and CGRA performance, for the fewest number of cycles and clusters

It should be noted here that these results are in no way intended to suggest that in most, or even many, cases Trimaran can perform on par with Macah and SPR. Both the Bayer and DWT mirroring kernels used here are not the main computation kernels for these applications. Instead, they are kernels selected particularly for their control-heavy properties. The Bayer kernel makes use of a conditional block with many mutually exclusive options, and the DWT kernel has a series of loops that are fundamentally serial. Both of these attributes are unusual, and it is reasonable to expect that for most application targeting a CGRA, the convolution kernel is much more typical. This kernel represents the computation heavy kernel that will frequently be the performance bottleneck for applications on Mosaic. Clearly these kernels are not well suited to Trimaran, but most multi-kernel applications will also have some kernels like the other two, where Trimaran could provide significant improvement. Generally, the control heavy kernels can be run with Trimaran, while SPR should be used for the primary computation task with a high degree of data-parallelism. If the application does not have any data-parallel kernels that perform significantly better in SPR, a CGRA may not be the correct target platform.

Given that some kernels do run well in Trimaran, and other kernels may not perform as well but may not be performance critical, it is appropriate to examine the performance per cluster of a Trimaran kernel. Significant energy and area savings can be achieved due to Trimaran using only one cluster for each kernel, and as Table 10 shows the performance loss is not always significant. This data can be found in Table 11. For this table, the number of clusters that consumes the fewest total issue slots from Figure 27 is used. The best performance per cluster tends to occur with fewer clusters than the best overall performance. One possible explanation is that there are diminishing returns from adding more hardware resources to a problem if there is no more parallelism available for SPR to exploit. Alternatively, the increased communication delay between more distant clusters may limit the benefits of using their resources at all.

Benchmark	Trimaran	Clusters	CGRA Total	Runtime
Convolution	7,214,477	4	636,804	11.3 x
Bayer Horizontal	519,520	2	274,576	1.89 x
Optimized Bayer	83,474	2	274,576	0.304 x
DWT Left Vertical Mirror	178,681	4	840,696	0.213 x

Table 11. Performance of Trimaran and CGRA execution scaled to the number of clusters used

Weighing the CGRA results by the number of clusters used paints a much better picture for Trimaran. These numbers are only meaningful if CGRA resources are scarce, in which case this comparison can give some idea of performance per cluster dedicated to the kernel. For the convolution, Macah and SPR on 4 clusters are still much faster than Trimaran (about 45.3 times) but this is only 11.3 times faster per cluster. Obviously, for parallel computation heavy kernels like this, the CGRA execution model is more efficient no matter what metric is used. The picture changes a little with the Bayer filter kernel, where the direct execution of the Macah-to-Trimaran C is only 1.89 times more efficient on a per cluster basis, and the optimized version runs much faster on Trimaran. This result would be even more skewed towards Trimaran, but the CGRA version only requires twice the hardware resources. The result is even more extreme in the DWT case, where CGRA execution is most efficient on 4 clusters. In this case, Trimaran is 4.7 times

more efficient per cluster. This result is probably due to the very high level of wasted parallel work that the DWT mirroring kernel requires under Macah and SPR.

Overall, these results show that in cases where Trimaran is well-suited for the kernel's control-heavy work load, the work done by a Trimaran cluster compared to one running part of a CGRA kernel is greater than the speedup number might indicate. It is important not to confuse these numbers with the actual performance relationship when the kernel is run on many clusters. It is simply a way of measuring the potential cluster utilization advantage of Trimaran execution. However, in the case where many independent data streams are being processed these numbers could be translated into performance gains simply by instantiating the Trimaran kernel many times.

7. Conclusion and Future Work

The CGRA execution model, as implemented in Mosaic 1.0, has been shown to provide very energy efficient execution for a large number of parallel applications. However, its strict modulo schedule and lack of control flow, other than predication, create a class of control-heavy applications where performance suffers significantly. This work suggests an alternative C compilation flow based on the Trimaran compiler to address these cases. A tool flow that maps Trimaran programs to a single-cycle custom Trimaran processor, built out of Mosaic components, demonstrates the feasibility of emitting Verilog configuration code from Trimaran.

This work also proposes a consensus architecture capable of executing both CGRA and Trimaran mode kernels. This architecture is designed to have reasonable Trimaran performance without incurring any performance penalty in CGRA mode. Additionally, the area and energy overhead was minimized as much as possible during the design. This resulted in a feasible architecture that was capable of single-cycle performance for most Trimaran instructions and required no more than two cycles to execute any instruction.

Although the consensus architecture was found to take more than 80 times as long to execute a highly parallel kernel in Trimaran mode, when compared to CGRA mode, other more control-heavy kernels run as fast or faster using Trimaran. Furthermore, Trimaran is significantly more efficient when it comes to Mosaic clusters consumed, since all kernels execute on a single cluster. Given the metric of total issue slots spent computing across all clusters, Trimaran is as much as 4.7 times as efficient for a control-heavy serial kernel. This suggests that a consensus architecture which provides a best of both worlds scenario in which each kernel runs under the execution model best suited for it could provide significant performance and energy gains over the current CGRA mode only Mosaic architecture and tool flow.

Given these results, users of Mosaic may find Trimaran compilation advantageous in any control-heavy scenario. Code that has large conditional blocks will tend to fall into this category. Applications with many small conditional blocks will also fit this model, whether these blocks are formed by a `case` statement, a standard `if` statement, or many

nested loops. In addition to these cases, code with less control that is not performance critical can also benefit from Trimaran execution. This is illustrated to some degree by the issue slots metric, but could be extended further to any kernel off the critical path. These kernels can all be executed on a single cluster in Trimaran mode, which will leave more clusters free for parallel execution of other kernels as well as potentially consuming less energy. Depending on the sophistication of the Mosaic floor planner, this may prove to be a better solution than relying on it to limit these kernels to a single cluster when compiled using Macah and SPR.

Future work on this project could include implementations of the consensus architecture and optimizations to Trimaran's scheduling algorithms to take advantage of execution on that architecture. Furthermore, the consensus architecture could be made more efficient if the Trimaran compiler was able to make use of the additional registers available in the fabric. This work sufficiently demonstrates the feasibility and performance benefits of the hybrid approach, as well as providing a design outline; however, further characterization of the energy and area overhead of the consensus architecture would be required before a specific implementation can be settled on.

References

[1] David Manners. (2010, May) FPGA Market Soaring To $4bn In 2010, says Gavrielov. Article. [Online]. http://www.electronicsweekly.com/Articles/19/05/2010/48677/FPGA-Market-Soaring-To-4bn-In-2010-says-Gavrielov.htm

[2] Infiniti Research Limited. (2011, March) Business Wire. [Online]. http://www.businesswire.com/news/home/20110401005410/en/Research-Markets-Global-Field-Programmable-Gate-Array-FPGA

[3] Cao Liang and Xinming Huang, "SmartCell: A power-efficient reconfigurable architecture for data streaming applications," in *IEEE Workshop on Signal Processing Systems, 2008*, Washington, DC, 2008, pp. 257-262.

[4] Altera. (2011, April) Accelerating DSP Designs with the Total 28-nm DSP Portfolio. White Paper. [Online]. www.altera.com/literature/wp/wp-01136-stxv-dsp-portfolio.pdf

[5] Xilinx. (2011, October) 7 Series DSP48E1 Slice. User Guide. [Online]. http://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

[6] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J. Kurdahi, Nader Bagherzadeh, and Eliseu M. C. Filho, "MorphoSys: An Integrated Reconfigurable System for Data-Parallel Computation-Intensive Applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465-481, 2000.

[7] Francisco Barat, Murali Jayapala, Tom Vander Aa, Rudy Lauwereins, Geert Deconinck, and Henk Corporaal, "Low Power Coarse-Grained Reconfigurable Instruction Set Processor," *Field Programmable Logic and Application*, vol. 2778, pp. 230-239, 2003.

[8] Mike Butts, "Synchronization through Communication in a Massively Parallel Processor Array," *IEEE Micro*, vol. 27, no. 5, pp. 32-40, Sept.-Oct. 2007.

[9] UW Embedded Research Group. (2006, June) Mosaic. [Online]. http://www.cs.washington.edu/research/lis/mosaic/index.shtml

[10] B. Van Essen et al., "Static versus scheduled interconnect in Coarse-Grained Reconfigurable Arrays," in *International Conference on Field Programmable Logic and Applications, 2009*, Prague, 2009, pp. 268-275.

[11] Brian Van Essen, "Improving the Energy Efficiency of Coarse-Grained

Reconfigurable Arrays," University of Washington, Seattle, Ph.D. Thesis 2010.

[12] B. Ylvisaker et al., "Macah: A "C-Level" Language for Programming Kernels on Coprocessor Accelerators," University of Washington, Seattle, Technical Report 2008.

[13] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "SPR: An Architecture-Adaptive CGRA Mapping Tool," in *ACM/SIGDA Symposium on Field-Programmable Gate Arrays*, 2009, pp. 191-200.

[14] Stephen Friedman, "Resource Sharing in Modulo-Scheduled Reconfigurable Architectures," University of Washington, Seattle, PhD Thesis 2011.

[15] Robin Panda and Scott Hauck, "Scheduled and Dynamic Communication in a Coarse Grained Reconfigurable Array," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2011.

[16] Lakshmi N. Chakrapani, John Gyllenhaal, Wen-mei W. Hwu, Scott A. Mahlke, Krishna V. Palem, and Rodric M. Rabbah, "Trimaran: An Infrastructure for Research in Instruction-Level Parallelism," *Lecture Notes in Computer Science*, vol. 3602, pp. 32-41, 2005.

[17] Trimaran. (2007, December) Trimaran: A Compiler and Simulator for Research on Embedded and EPIC Architectures. PDF. [Online]. http://www.trimaran.org/docs/trimaran4_manual.pdf

[18] Yogesh Chobe, Bhagi Narahari, Rahul Simha, and Weng-Fai Wong, "Tritanium: Augmenting the Trimaran Compiler Infrastructure To Support IA-64 Code Generation," The George Washington University, Washington DC, EPIC-1 Workshop 2001.

[19] L.N. Chakrapani, W.F. Wong, and K.V. Palem, "TRICEPS: Enhancing the Trimaran Compiler Infrastructure To Support StrongARM Code Generation," Georgia Institute of Technology, Atlanta, Technical Report 2001.

[20] Ming Yan, Ziyu Yang, Liu Yang, Lei Liu, and Sikun Li, "Practical and Effective Domain-Specific Function Unit Design for CGRA," *Lecture Notes in Computer Science*, vol. 6786, pp. 577-592, 2011.

[21] Bhuvan Middha, Anup Gangwar, Anshul Kumar, M. Balakrishnan, and Paolo Ienne, "A Trimaran based framework for exploring the design space of VLIW ASIPs with coarse grain functional units," in *Proceedings of the 15th international symposium on System Synthesis*, Kyoto, 2002, pp. 2-7.

[22] Steven M. Rubin. (2010, December) Electric User's Manual. [Online]. http://www.staticfreesoft.com/jmanual/

[23] James C. Dehnert, Peter Y.-T. Hsu, and Joseph P. Bratt, "Overlapped loop support in the Cydra 5," in *ASPLOS-III Proceedings of the third international conference on Architectural support for programming languages and operating systems*, New York, 1989, pp. 26-38.

[24] Vinod Kathail, Michael S. Schlansker, and B. Ramakrishna Rau, "HPL-PD Architecture Specification: Version 1.1," Hewlet-Packard Compiler and Architecture Research, Technical Report HPL-93-80, 2000.

[25] Bryce E Bayer, "Color imaging array," 3971065, July 20, 1976.

[26] Ingrid Daubechies, "The Wavelet Transform, Time-Frequency Localization and Signal Analysis," *IEEE Transactions on Information Theory*, vol. 36, no. 5, pp. 961-1005, September 1990.

[27] Tom Fry, "Hyperspectral Image Compression on Reconfigurable Platforms," University of Washington, Seattle, Master's Thesis 2001.

Appendix A

2D Convolution Macah Source Code

```
task filler (in_port inStrm, out_port midStrm) {

  int i,j,k,l;
  int strip[K_S+1][Z_W];
  int result;
  int conv_kern[K_S][K_S];
  int sliding_win[K_S][K_S+1];

  for(i=0;i<K_S;i++){
  for(j=0;j<K_S;j++){
        conv_kern[i][j]=i+j;
  }
  }

  trimaranKernelStart();
  kernel fillk {
  for(i=0;i<Z_H;i++){
    for(j=0;j<Z_W;j++){
      strip[K_S][j] <? inStrm;
      result=0;
      FOR(k=0;k<K_S;k++){
        sliding_win[k][K_S]=strip[k+1][j];
        FOR(l=0;l<K_S;l++){
        result+=sliding_win[k][l+1]*conv_kern[k][l];
        sliding_win[k][l]=sliding_win[k][l+1];
        }
        strip[k][j]=strip[k+1][j];
      }
      midStrm <! result;
    }
  }
  trimaranKernelEnd();
}
```

Bayer Filter Macah Source Code

```
task filler (in_port inStrm, out_port midStrm) {

  int tmpRow[2];
  int col, row;
  int tmp;

  trimaranKernelStart();
  kernel fillk {
    for (row = -1; row <= HEIGHT; row++) {
      for (col = -1; col <= WIDTH; col++) {
        //remember first two collumns
        if (col == -1) {
          tmpRow[0] <? inStrm;
          tmpRow[1] <? inStrm;
          midStrm <! tmpRow[1];
        //don't skip col 0 and last
```

```
          } else if (col == 0 || col == WIDTH) {
            midStrm <! tmpRow[0];
          //repeat col 1 and second to last
          } else if (col == 1 || col == (WIDTH - 1)) {
            midStrm <! tmpRow[1];
          //remember last two collumns
          } else if (col == (WIDTH - 2)) {
            tmpRow[0] <? inStrm;
            tmpRow[1] <? inStrm;
            midStrm <! tmpRow[0];
          //normal collumns
          } else {
            tmp <? inStrm;
            midStrm <! tmp;
          }
        }
      }
    }
    trimaranKernelEnd();
  }
```

Discrete Wavelet Transform Macah Source Code

```
  task filler (in_port inStrm, out_port midStrm) {
    int h;
    int i;
    int j;

    int offset;

    int data[MAX_COEF_COUNT * HALF_STRIPE_WIDTH];

    trimaranKernelStart();
    kernel fillk {
    for (h = 0; h < SIZE / STRIPE_WIDTH; h++)
    {
      //read in necessary mirror data
      for (i = 0; i < (MAX_COEF_COUNT * HALF_STRIPE_WIDTH); i++)
      {
        data[i] <? inStrm;
      }

      //write mirrored data [3 2 | 1 2 3...]
      for (i = MAX_COEF_COUNT - 1; i > 0; i--)
      {
        offset = i * HALF_STRIPE_WIDTH;

        for (j = 0; j < HALF_STRIPE_WIDTH; j++)
        {
          midStrm <! data[offset + j];
        }
      }

      for (i = 0; i < (MAX_COEF_COUNT * HALF_STRIPE_WIDTH); i++)
      {
        midStrm <! data[i];
      }
```

```
        //write standard data
        for (i = 0; i < (SIZE - MAX_COEF_COUNT) * HALF_STRIPE_WIDTH;
i++)
        {
          data[0] <? inStrm;

          midStrm <! data[0];
        }
      }
      }
      trimaranKernelEnd();
    }
```

Bayer Filter Optimized Trimaran Source Code

```
int main (int argc, char *argv[])
{
  volatile int inStrm ;
  volatile int midStrm ;
  int tmpRow[2] ;
  int col ;
  int row ;
  int tmp ;
        for (row = -1; row <= HEIGHT; row++) {
          tmpRow[0] = inStrm;
          tmpRow[1] = inStrm;
          midStrm = tmpRow[1];
          midStrm = tmpRow[0];
          midStrm = tmpRow[1];
          for (col = 2; col < (WIDTH - 2); col++) {
              tmp = inStrm;
              midStrm = tmp;
          }
          tmpRow[0] = inStrm;
          tmpRow[1] = inStrm;
          midStrm = tmpRow[0];

          midStrm = tmpRow[1];
          midStrm = tmpRow[0];
        }
}
```