Enhanced Loop Flattening for Software Pipelining of Arbitrary Loop Nests

Author information withheld for blind review

June 11, 2010

Abstract

This paper introduces and evaluates *enhanced loop flattening*, a compiler framework for transforming an arbitrary set of nested and sequenced loops into a single loop with additional logic to implement the control flow of the original code. Loop flattening allows conventional software pipelining algorithms to effectively pipeline nested loops. Conventional software pipelining approaches are only applicable to inner loops, which can result in poor resource utilization.

Existing approaches to loop flattening tend to create long latency inter-iteration feedback paths, which limit the performance of pipelined loops. Enhanced loop flattening gives the compiler the ability to precisely control iteration distances along each control flow path, reducing the initiation interval (the "clock period" of a pipelined loop) by inserting "pipeline bubbles" under specific dynamic conditions. On a benchmark set of compute-intensive kernels, we found that conventional loop flattening performed somewhat worse than inner-loop only pipelining, but enhanced loop flattening with a simple heuristic for iteration distance selection performed substantially better (geometric mean of 30% less run time).

Because of the fairly large number of single-bit operations and registers that are required for predicate logic, enhanced loop flattening is most directly applicable to FPGA-like parallel accelerators.

1 Introduction

This paper describes a new method for *flattening* a complex loop nest into a single loop. Conditional guards are used to implement the original control flow and select operations to implement the original data flow. This transformation can be seen as a generalization of if-conversion—which is only applicable to acyclic control flow—to arbitrary cyclic control flow. The primary benefit of flattening that we explore is that the flattened loop can be software pipelined with conventional algorithms like iterative modulo scheduling [1] or swing modulo scheduling [2].

Software pipelining is a compiler technique for converting loop-level parallelism into instruction-level parallelism. It is particularly important for compiling to fine-grained massively parallel architectures, like field-programmable gate arrays (FPGAs), graphics processing units (GPUs) and very long instruction word (VLIW) processors. The methods described in this paper are directly applicable to FPGA-like architectures; we believe they could be extended to work on other kinds of machines.

Many different variants of pipelining[1] exist; most require that the input to the pipelining algorithm be the body of a single loop with no function calls or non-trivial control flow. Conventional inner-loop-only pipelining (ILOP) can introduce considerable inefficiency in applications with nested loops. Flattening allows for pipelining around outer loops as well as inner loops, which can lead to better efficiency. However, conventional loop flattening algorithms can introduce unnecessary long-latency feedback paths that limit the effectiveness of pipelining. In this paper we describe *enhanced loop flattening*, a method for flattening arbitrary cyclic control flow (not just reducible loops) that produces efficiently pipelinable code.

2 Background

Software pipelining is a family of compiler methods for scheduling and resource allocation that exploit the fact that while the number of parallel operations available in a single iteration of a loop is often limited, operations from later iterations can be executed before earlier iterations have completed as long as all dependencies are respected. The result of applying pipelining to a loop is a *steady state* schedule as well as a prologue and epilogue. The basic terminology of pipelining is

[1] In this paper we use "software pipelining" and "pipelining" interchangeably.

illustrated in figure 1. We will use the following abbreviations (which are more completely defined in succeeding paragraphs) throughout the paper.

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

In the pipelining literature, throughput is usually discussed in terms of the *initiation interval* (II), which is the amount of time between initiation of consecutive iterations of the loop. Lower II is generally better. In the best case, a pipelined loop can run at a throughput of one loop iteration per cycle. Each iteration will likely have a latency higher than one cycle, but because many iterations are overlapped, the throughput can be much higher than the inverse of the latency.

There are two important limitations on the achievable II: resource limits and feedback dependencies. If a loop body requires N of some resource, but there are only M ($<N$) available in the architecture, the II must be at least N/M. Perhaps less obviously, if some static operation depends (directly or indirectly) on an earlier execution of itself, the latency of the dependence chain between those two executions also limits the II. In figure 1 the highlighted inter-iteration feedback path has a latency of 5 (assuming unit latency for all operations along the path) and an iteration distance of 2, which means the II can be no lower than $5/2 = 2.5$ time units. In most cases we measure time units in machine cycles, which means that fractional IIs cannot be implemented.

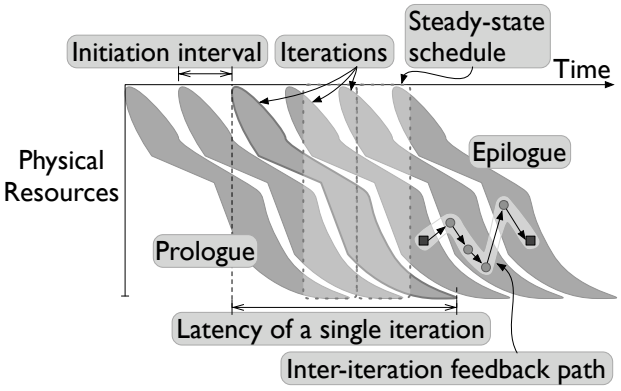

Figure 1: The basic terminology of loop pipelining. The repeated shape of the loop iterations signifies that in conventional software pipelining all iterations share the same schedule and resource assignments.

[describe *criticality*, maybe]

In the context of the massively parallel architectures we are interested in, computational resources are so abundant that for most applications we are more concerned about the minimum II

imposed by feedback paths (the RecII) than resource limits (the ResII). Because of the importance of the RecII, most of the engineering we describe in this paper is aimed at avoiding long-latency inter-iteration feedback paths.

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

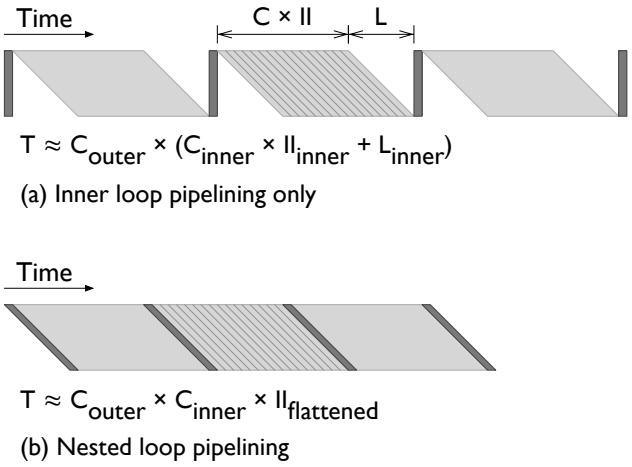

Figure 2: Sketches of the execution of a nested loop. In case (a), only the inner loop is pipelined, which results in the empty triangles at the beginning and end of each iteration of the outer loop. In case (b), the whole loop nest is pipelined together, which improves performance roughly in proportion to the ratio of $C_{inner} \times II$ to L.

For loops with very high trip counts, the latency of a single iteration is not usually important for overall program performance. However, the lower the trip count, the more important *prologue* and *epilogue* periods are. During prologue and epilogue, the hardware resources are underutilized, which negatively impacts performance. As illustrated in figure 2, if the prologue and epilogue periods of an inner loop can be overlapped properly, the total execution time will be approximately $C_{outer} \times C_{inner} \times II$. On the other hand, If the prologue and epilogue cannot be overlapped, the total execution time will be approximately $C_{outer} \times (L_{inner} + C_{inner} \times II)$.

2.1 Hierarchical Reduction

One existing approach to pipelining loops with complex control flow is *hierarchical reduction*[3]. The hierarchical reduction (HR) method applies pipelining to more deeply nested blocks of code (like branches of an if/then/else or inner loop bodies), then treats the resulting schedule and resource allocation as a "complex primitive" and applies pipelining again to the next level. HR has two

weaknesses compared to the method proposed in this paper:

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

Our compiler does not support the complex primitive concept required to implement HR, so we cannot experimentally evaluate the cost of solving the scheduling problem hierarchically. However, the authors of [4] compared HR to predication in the context of if-conversion, and found that HR was significantly less efficient.

2.2 Flattening

Another approach to pipelining nested loops is to first apply *loop flattening* [5, 6], which merges all the blocks of a loop nest into a single loop body with additional control logic to regulate when the various blocks should execute and how values flow between blocks. The usual assumption in loop flattening is that each block can execute at most once in a single iteration of the flattened loop; flattening does not duplicate any blocks. Conventional approaches to flattening are greedy in the sense that they try to execute each block as soon as possible as measured by the iteration count of the flattened loop. The greedy approach minimizes the total trip count of the flattened loop, but it can create long inter-iteration feedback paths. Enhanced loop flattening strategically increases the iteration distance along certain control flow paths. In other words, some blocks execute in a later iteration than is strictly necessary. This iteration distance increase is like inserting "pipeline bubbles" that can reduce the RecII by increasing the iteration distance along inter-iteration feedback paths. However, increasing iteration distances can also inflate the trip count, so we want to increase iteration distances just enough to reduce the II.

For readers who have other favorite loop optimizations, like unrolling or fusion, we note that pipelining is a complement to those other tools, not a replacement for them. For example, if a program has two sequenced loops with similar trip counts and no true dependences between them, it is probably best to apply fusion first. Whatever other optimizations are applied, it is often still beneficial to use pipelining to perform the final scheduling.

Flattening out control flow (with if-conversion or loop flattening) has the important side-effect that all operations are executed, whether their results are needed or not. The unnecessary executions can be a performance problem, which is discussed further in section 5.

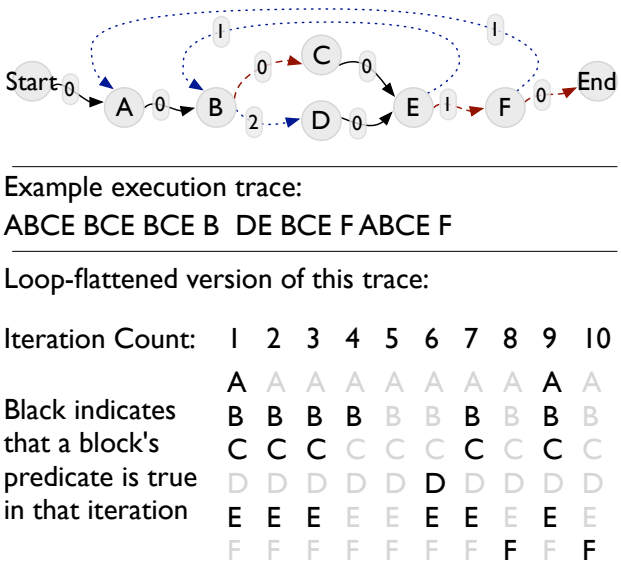

Figure 3: An example CFG with iteration-distance annotations, an example trace through that CFG, and a view of how that trace would execute in a loop-flattened version of the CFG. The specific iteration distance annotations were chosen for illustrative purposes, and do not necessarily represent the best choices for this graph by any performance metric.

3 Enhanced Loop Flattening

Enhanced loop flattening translates a control flow graph (CFG) representation of a loop nest into a single dataflow graph (DFG) that can be scheduled with conventional software pipelining algorithms. The output DFG can be thought of as the body of a single loop where regions that correspond to different parts of the original loop nest are guarded by different predicates. The translation process generates the necessary predicate logic and select operations to implement the control flow and data flow of the program. In each iteration of the flattened loop exactly one of the regions of the DFG is active. Our novel contributions are:

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

An overview of the enhanced loop flattening method:

1. Annotate CFG edges with iteration distances based on profiling information or a static heuristic.

2. Visit the CFG nodes (basic blocks) in an intra-iteration topological order. For each basic block:

   (a) Generate the logic for its predicate

   (b) Generate select operations to choose values for each variable from the last producer in each possible predecessor basic block.

First we decide what the iteration distance should be along each edge in the CFG. Iteration distances are represented as non-negative integer labels on edges. When the flattened loop makes a zero-distance control transition, the source and target blocks execute in the same iteration; when the distance is d (> 0), the execution goes around the loop d times before executing the target block. We can think of this as putting d bubbles into the software pipeline.

Figure 3 shows a control flow graph that has been annotated with iteration distances. The "flattened trace" at the bottom shows how an example trace maps to iterations of the flattened loop. Notice that after the execution of block B in iteration 4 there are two "bubble iterations" before block D executes. This corresponds to the "2" annotation on the B-D edge in the CFG.

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

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

Two pieces of information need to be known or estimated in order to optimize iteration distances: the relative execution frequencies of the CFG edges, and the criticality of the cycles. It is generally better to increase iteration distances on edges with lower execution frequencies, because that will inflate the trip count less. Also it is generally better to increase iteration distances on edges that

participate in more critical feedback paths, because that is more likely to reduce the II.

Execution frequency information can be collected by profiling. Cycle criticality information can be found by iteratively performing loop flattening and scheduling (which comes later in the compilation process). However, our compiler does not currently have support for either of these techniques, so we use a static heuristic based on the loop structure of the code. The heuristic is described in more detail in the evaluation section.

3.1 Intra- and Inter-Iteration

In this paper we use "intra-iteration" variants of many flow graph concepts like paths, dominance and topological order. Edges with a distance of zero are intra-iteration, and edges with a higher distance are inter-iteration. Informally, the intra-iteration variant of a concept is simply the conventional concept on a graph where all the inter-iteration edges have been removed. Here are more formal definitions of the important relations reachablility, dominance and post-dominance.

x reach y	There is an intra-iteration path from x to y.
x dom y	Every intra-iteration path from either the start node or an inter-iteration edge to y includes x.
x pdom y	Every intra-iteration path from y to either the end node or an inter-iteration edge includes x.

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

3.2 Basic Blocks

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

means that producer and consumer operations are directly connected to each other.

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

3.3 Predicates

Every basic block has a predicate that is true on iterations of the flattened loop when the block should execute and false on iterations when the block should not execute. We extend the notion of predicates to CFG edges by defining the predicate of an edge to be the predicate of its source block logically ANDed with the condition under which that block takes a branch along that edge. Edges whose source node has a single successor (i.e. an unconditional branch) have a predicate that is identical to the source node.

Our method for computing predicates is illustrated in figure 4. It is similar to existing methods for if-conversion [7], which in turn are based on the notion of control dependence from the program dependence graph (PDG) literature. To calculate the predicate for a CFG node x, we find its post-dominated region, the set $\{y | x \ \mathsf{pdom} \ y\}$. Intuitively, once one of the nodes in x's post-dominated region executes, we can be sure that x will execute in the current iteration.

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

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

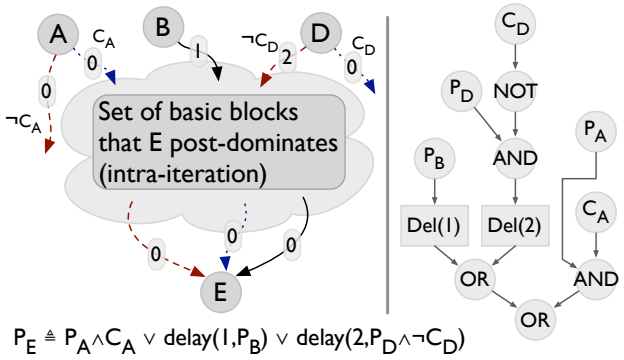

Figure 4: A piece of a control flow graph on the left, with the predicate logic for block E represented as a formula and a data flow graph. P_x means "predicate for block x", C_x means "branch condition for exiting block x", and $Del(N)$ means N-iteration delay.

As an easily computed estimate of criticality for each control dependence edge, we count the smallest total iteration distance on any cyclic path backwards to the block for which we are generating a predicate. If there is no such path, we consider the distance for that edge to be infinite. Criticality is taken to be the inverse of distance. This heuristic aims to keep latencies around the inner loops as low as possible.

3.4 Select Operations

For basic blocks with multiple predecessors, there can be multiple producer operations for each variable. On each iteration the dynamic control flow determines which value should be seen by consumer operations. In some implementations of if-conversion this choice is implemented with predicated writes to a global register file. However, even for conventional architectures this approach introduces performance challenges related to register renaming [8], and massively parallel architectures have no global register file at all. Therefore, we use the other common method for controlling the data flow, which is select operations with explicit control inputs.

The logic for inserting select operations is similar in some ways to predicate generation logic. However, the fundamental difference between the two is that while predicates must be either true or false on every iteration according to the control flow of the program, on iterations when the consumer block of a select operation does not execute it does not matter whether the control input

is true or false. We can exploit these *don't care* cases to substantially improve the size and latency of the select trees we generate.

Select control logic for a *consumer* block can be generated locally by using the predicates of the direct incoming edges. However, those predicates encode many branch decisions (from the current iteration as well as previous iterations) that might not be essential for selecting the correct values for *consumer*. The A-B-D-E subgraph in figure 5 helps illustrate this point. The selects for block E could be controlled by the B-E edge predicate, with the value from the producer in block B on the *then* input and value from the producer in block D on the *else* input. Alternatively, the D-E predicate could be used, with the data inputs swapped. However, the B-E and D-E edge predicates include logic for the B and D branches, respectively, and if the program takes a branch that does not lead to E, we simply don't care what data is selected. So in the simplified A-B-D-E graph, the best choice for selecting inputs for E is the branch condition for block A. Note that the predicate for A is also not essential, because if A does not execute E will certainly not execute.

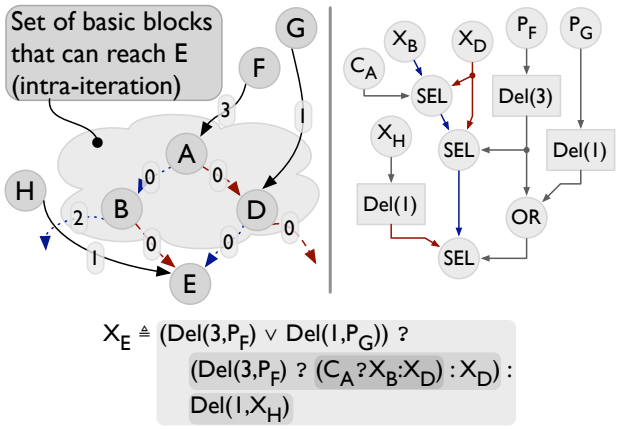

Figure 5: A piece of a control flow graph on the left, with the select logic for the incoming value of variable X to block E represented as a formula and a data flow graph. P, C, and Del have the same meaning as in figure 4. X_y means the last definition of the variable X in block y. The "`?:`" expressions have the same meaning they do in C/C++/Java.

The algorithm for generating select operations for some *consumer* block uses two edge labels:

- P_e, where P stands for "producer" and e is a reference to one of *consumer*'s incoming edges.
- $B(C, \langle e_1, l_1\rangle, \ldots, \langle e_n, l_n\rangle)$, where B stands for "branch", C is a reference to a specific branch condition, e_x is a reference to an edge and l_x is a label. The order of the edges matters in a B label. If C is a Boolean branch, the first edge is "then" and the second is "else".

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

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

1. Label each of *consumer*'s direct incoming edges, e, with P_e.

2. Visit the nodes that can reach *consumer* in reverse topological order. For each node n let $S_n = \{e \mid (source(e) = n) \wedge (e \text{ reach } consumer)\}$. If all edges in S_n have the same label, label all of n's incoming edges (intra- and inter-iteration) with that label. If not all of the labels are the same, label all incoming edges with $B(C, \langle e_1, l_1\rangle, \ldots, \langle e_n, l_n\rangle)$, where C is n's branch condition (n must end in a branch to have multiple outgoing edges), and the list of edges are S_n, together with their labels.

After running this algorithm, all inter-iteration edges whose destination can reach *consumer* will have a label. We now have to consider two cases: either all these inter-iteration edge labels match or they do not. If all labels match we can generate a select tree for each variable by following the label with this intra-iteration select algorithm: P_e means choose the last definition of the variable from e's source basic block. If e is an inter-iteration edge, we add a d-iteration delay to this connection. In contrast to the delays in the predicate logic, these delays do not need to be initialized, because the value of the output does not matter unless the predecessor block is actually executed in an earlier iteration.

$B(C, \langle e_1, l_1\rangle, \ldots, \langle e_n, l_n\rangle)$, means create a select operation that is controlled by C and takes as data inputs the result of recursively running the intra-iteration select algorithm on each of the sub-labels $l_1, \ldots, l_n$. Decomposing multi-way selects generated from `switch` statements or CFG transformations into trees or chains of binary selects is similar to the inter-iteration select control problem discussed below.

It is common for the last producer for a particular variable to be the same along all control paths (e.g. when some variable is not modified on either branch of an if-then-else). We leave out the select operation entirely in this case as an early optimization. It is simple to remove selects with identical inputs during a later pass, but doing it that way can create a huge number of selects in the intermediate representation.

3.4.1 Inter-iteration Select Control

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

We use a heuristic for inter-iteration select logic generation that is illustrated in figure 6. Like the predicate logic OR chains, we compute an estimate of the criticality of each of the paths, then make a chain of selects with the most critical select "closest" to the consumer and the least critical select "farthest". Select trees can be further optimized with more detailed criticality information, but that is beyond the scope of this paper.

To control the select chain, we need some expression that represents a particular edge executing; we will call this exec_e. The least critical select operation is controlled by exec_e for the least critical edge; if exec_e is true, select along the least critical path, if it is false, select along the next-least critical path. The next select in the chain is controlled by the OR of the two least critical exec_e expressions, and so on. This organization has the important benefit that exec_e for the most critical edge is not part of the logic at all. Imagine a simple nested loop where the inner loop has two incoming edges: the loop-back and the external entry. The select operations for the inner loop should be controlled by the expression that represents that the loop is starting, so that they do not directly depend on the loop completion logic for the inner loop itself.

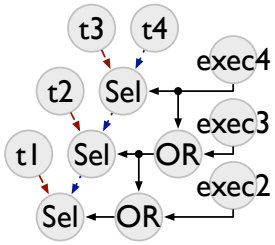

Figure 6: An example select chain for the inter-iteration part of data flow control. "t_n" represents the mux tree generated from intra-iteration select label "l_n". t_1 is the most critical; t_4 the least. exec_n is an expression that represents a predecessor block executing in an earlier iteration. Notice that the most critical "exec" is not part of the logic at all.

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

cases to be exploited in select logic generation.

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

3.5 Implementation Issues

The input CFG to loop flattening can have arbitrary structure, include irreducible control flow created by GOTOs or other optimizations. We cannot handle dynamic non-local control transfers like exceptions and continuations, though in some cases they can be converted to static control flow. Our target architectures do not support function calls, so our compiler currently inlines all function calls, though flattening itself has no issues with supporting function calls.

The outputs of basic blocks are initially represented by stubs that get replaced with connections to real operations after a block is processed. Having these stubs is necessary because in general CFGs have cycles, so some blocks will need to be processed before all of their predecessors have been processed.

The pseudo-data flow scheduling constraints for keeping side-effecting operations in program order can sometimes be relaxed. For example multiple sequential reads from the same array without any intervening writes do not need to execute in any particular order.

3.6 Scheduling, Placement and Routing

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

For applications with non-trivial loop nesting, the predicate logic and select trees created by our flattening method can be sufficiently complex that we expect targeting conventional processors would not work very well. In particular a large number of predicate values have to be maintained and pipelined. This would create severe register pressure and would be an inefficient use of wide registers for Boolean values. In [9] the authors observe a similar problem with predication, and propose architectural features to address the problem.

Our current target architectures are FPGA-like in the sense that they have large amounts of compute resources compared to conventional VLIWs, support deep pipelining of data values well, and support a modest amount of single-bit computation well. Because these architectures are spatial, our back-end includes not only the scheduling and resource allocation of conventional software pipelining, but also temporospatial placement and routing, like that described in [10, 11].

4 Evaluation

In the context of C-like languages for parallel accelerators, the most important advantage of enhanced loop flattening over plain software pipelining is that applications that have more complex control flow can be written in a natural style. Existing systems for translating from sequential languages to highly fine-grained parallel architectures, like Impulse-C [12] and StreamC/KernelC [13] force the programmer to settle for inner-loop-only pipelining (ILOP) or essentially do the flattening transformations by hand, which is hard to get right and results in messy, unmaintainable code.

To quantify the benefits of enhanced loop flattening, we compare the run times of a set of benchmarks compiled with enhanced loop flattening in a number of different experimental configurations. Our target architecture is a simulated accelerator with 200 ALUs arranged in clusters with a grid interconnect like that described in [14].

All run times are normalized to the ILOP implementation. We implement ILOP in our infrastructure by adding extra scheduling constraints that prevent the overlap of operations from different blocks, except the overlap of inner loop operations with each other. We then add enough iteration distance on outer loop connections to cover the latency of those blocks.

The other point of comparison is conventional loop flattening, which we call greedy because blocks are scheduled in the flattened loop as soon as possible.

Enhanced loop flattening provides a flexible framework for setting iteration distances, but leaves open exactly what those distances should be. We implemented and evaluated a family of heuristics for setting iteration distances based on the loop structure of the program. We use the algorithm from [15] to identify the CFG nodes that participate in all loops, including arbitrary nesting of reducible and irreducible loops. Here we mean loop in the programming language sense, not the graph theoretic sense, so a simple diamond-shaped (or hammock-shaped) CFG with a back-edge from the bottom to the top is a single loop, not two.

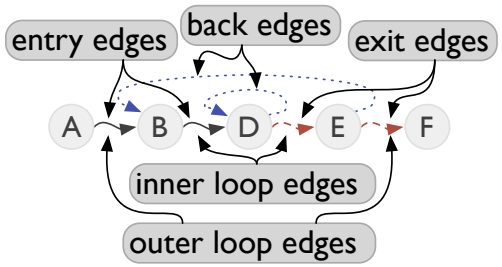

Figure 7: The family of static iteration distance heuristics use the loop structure of the program. Particular heuristics in the family put different iteration distances on edges based on whether they are entry or exit edges, and whether they are entering/exiting an inner or outer loop.

Once we have identified the loops, we set the iteration distance on all back-edges to one. Back-edges can be identified in the conventional way with a depth-first traversal of the nodes in each loop. Further, different members of the heuristic family set different additional edges to a distance of one. Figure 7 illustrates the different kinds of edges. In this setup, we can choose to increase the distance on all loop entry edges, only inner loop entry edges or no loop entry edges. We have the same set of choices for loop exit edges. If we choose to add distance to no entry or exit edges, we are back to the greedy loop flattening case.

Figure 8 shows the results for our set of 10 benchmark applications. The "Heuristic-A" bars show the runtimes when the iteration distances are set to one on all loop entry and exit edges. Three of the applications (cordic, firs, and ed) have only a single loop, so we see no difference at all across the different setups, as we would expect. For the rest of the applications, we see that the greedy flattening is almost always worse than ILOP. The enhanced loop flattened implementations with our simple heuristic for setting iteration distances are all at least as fast as ILOP, and some are substantially faster.

The heuristic enhanced loop flattening performs much better than greedy flattening, even though the trip counts are somewhat higher, because the initiation intervals for the greedy implementations are much higher. The "Optimistic" bars in figure 8 show the run times we would see if we could get the lower trip counts of greedy flattening and the lower IIs of enhanced flattening at the same time. This represents an optimistic bound on the performance improvement that could be achieved by setting the iteration distances better.

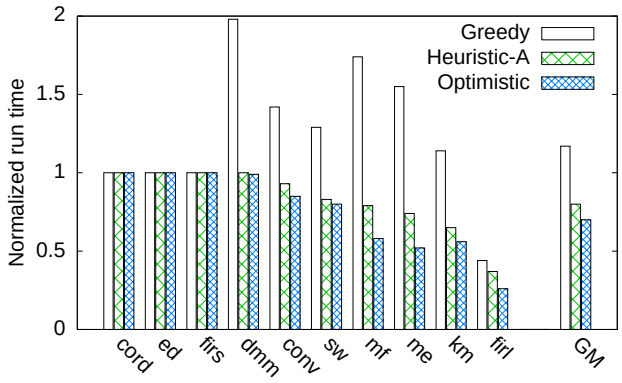

Figure 8: Normalized run times for a suite of applications compiled with enhanced loop flattening. The baseline is inner-loop only pipelining. "Greedy" is conventional loop flattening with non-zero iteration distances on back edges only. "Heuristic-A" is enhanced loop flattening with an iteration distance of 1 on all loop entry and exit edges. "GM" is geometric mean.

Enhanced loop flattening outperforms ILOP in proportion to the latency of the inner loops, and inversely proportionally to the trip counts of the inner loops. At one extreme, dense matrix multiplication (dmm) has almost identical performance. The reason for this is that we use a SUMMA-style algorithm where the inner loop is essentially many parallel multiply-accumulate loops. This means that the latency of the inner loop is very low and the benefit of pipelining around the outer loops is small as a consequence. At the other extreme, our banked FIR filter implementation (firl) shows a large performance difference, because there is a very long-latency chain of additions in the inner loop.

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

Table 1 shows the performance results broken up into initiation interval and trip count. We also include data for enhanced loop flattening with the iteration distance heuristic set to add distance to entry and exit edges for only the inner loops (Heur-I). The first column shows the natural IIs of

App	IIs				Trip Counts			Run Times		
	ILOP	Greedy	Heur-I	Heur-A	Greedy	Heur-I	Heur-A	Greedy	Heur-I	Heur-A
cord	2	1	1	1	1	1	1	1	1	1
ed	6	1	1	1	1	1	1	1	1	1
firs	2	1	1	1	1	1	1	1	1	1
mm	4	2.0	1	1	0.99	1.00	1.00	1.98	1.00	1.00
conv	3	1.67	1.33	1	0.85	0.93	0.93	1.42	1.23	0.93
sw	5	1.6	1	1	0.80	0.83	0.83	1.29	0.83	0.83
mf	3	3.0	1	1	0.56	0.74	0.79	1.74	0.74	0.79
me	4	3.0	1	1	0.52	0.69	0.74	1.55	0.69	0.74
km	4	2.0	1	1	0.57	0.65	0.65	1.14	0.65	0.65
firl	3	1.67	1	1	0.26	0.37	0.37	0.44	0.37	0.37
GM		1.66	1.03	1	0.71	0.79	0.80	1.17	0.81	0.80

Table 1: Performance data broken up into initiation interval and trip count. In addition to the greedy and heuristic with non-zero iteration distances on all entries and exits (Heur-A), we also show results for the static heuristic with non-zero iteration distances on only the inner loop entries and exits (Heur-I).

the inner loops of the applications, and the rest of the data is all normalized to the ILOP values. In the Trip Count section of the table, you can see that adding more iteration distance (Heur-A vs. Heur-I) increases the trip counts slightly. The Heur-A configuration achieves a slightly higher average performance than Heur-I because there is one application that is not able to get back down to the minimum II in the Heur-I configuration.

Our final experiments examined the value of the more sophisticated dominator-based inter-iteration select control mechanism described at the end of section 3.4.1. This optimization has no effect on trip count and only had a large enough impact to reduce the II in a very few cases, so we looked at a different quality metric that is a more precise measure of the "slack" in the inter-iteration feedback paths. For every operation that participates in a feedback path we compute how much additional latency that operation would need to have in order to make it II-critical (i.e. any more latency would cause the minimum RecII to increase). We use the table method from [1] to compute these numbers. We found that using the dominator-based select logic led to noticeably higher average slack, which means that those implementations were fractionally closer to achieving a lower II.

5 Discussion

Loop flattening and its acyclic cousin if-conversion share an important performance challenge, which is that all operations execute on every iteration, whether their results are needed or are going to be discarded. In loop nests with lots of infrequently executed operations this can lead to a lot of wasted execution resources. In the future we plan to address this problem by exploring ways that operations that are guaranteed to never execute in the same iteration of the flattened loop can share physical resources. This problem was addressed in the acyclic case by the authors of [16] (who mostly ignored spatial distribution issues). We believe that extending those results to incorporate spatial distribution and iteration distances is an interesting and tractable challenge.

To facilitate this sharing between operations that execute in mutually exclusive conditions, we build a *control dependence graph* (CDG)[17], which keeps track of the conditions under which each part of the DFG should actually execute. We are also considering a modified CDG that represents iteration distances between different branches.

5.1 Unbalanced Diamonds

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

This kind of selective increase of distances along specific infrequent, high latency control paths makes enhanced loop flattening a more dynamic approach to scheduling loops than conventional software pipelining. In contrast to fully dynamic dataflow systems, like [18] and [19], we still schedule the flattened loop body fully statically. The static approach has the benefit that we can statically set up execution resource sharing between mutually exclusive operations to avoid poor utilization. In a purely dynamically scheduled system, resource sharing requires dynamic arbitration, which is expensive enough that it is hard to actually benefit from sharing.

6 Conclusions

In this paper we presented enhanced loop flattening, which can be used to software pipeline nested loops with arbitrarily complex static control flow. The ability to seamlessly pipeline around inner and outer loops is most beneficial when the pipelining is very deep (equivalently, when the number of iterations overlapped in the steady state is high, or when the ratio of the latency of a single iteration to the initiation interval is high) and the trip counts of the inner loops are not extremely high. Enhanced loop flattening provides a flexible mechanism for increasing the iteration distance along specific control paths, which can be used to balance the competing performance factors of trip count and initiation interval of the flattened loop.

In order to avoid unnecessary inter-iteration feedback paths, we propsosed algorithms for predicate and select operation generation that use "intra-iteration" variants of the classic flow graph relations reachability, dominance and post-dominance and heuristics for minimizing inner loop latencies. We suspect that further improvement of these algorithms is possible, but in our experiments we did not find that they created any unnecessary II-limiting feedback paths.

Our implementation and the experiments we performed with it showed that even for relatively simple loop nests, pipelining the flattened loop has a better combination of trip count and initiation interval than inner loop only pipelining. We also showed that greedy loop flattening–with non-zero iteration distances on back edges only–creates long inter-iteration data flow feedback paths. Increasing iteration distances on less frequently executed control paths can improve performance by decreasing the initiation interval.

7 Acknowledgements

This work was supported by Department of Energy grant #DE-FG52-06NA27507, and NSF grant #CCF-0702621.

References

[1] B. R. Rau, "Iterative Modulo Scheduling: An Algorithm For Software Pipelining Loops," in *Proceedings of the 27th annual international symposium on Microarchitecture*. ACM Press,

1994.

[2] J. Llosa, A. González, E. Ayguadé, and M. Valero, "Swing modulo scheduling: A lifetime-sensitive approach," in *PACT '96: Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 1996.

[3] M. Lam, "Software pipelining: an effective scheduling technique for VLIW machines," in *ACM SIGPLAN conference on Programming Language design and Implementation*. ACM Press, 1988.

[4] N. J. Warter, S. A. Mahlke, W.-M. W. Hwu, and B. R. Rau, "Reverse if-conversion," in *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*. ACM, 1993.

[5] A. M. Ghuloum and A. L. Fisher, "Flattening and parallelizing irregular, recurrent loop nests," *SIGPLAN Not.*, vol. 30, no. 8, pp. 58–67, 1995.

[6] P. M. Knijnenburg, "Flattening VLIW code generation for imperfectly nested loops," Department of Computer Science, Leiden University, Tech. Rep., 1998.

[7] W. Chuang, B. Calder, and J. Ferrante, "Phi-predication for light-weight if-conversion," in *CGO '03: Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 2003.

[8] P. H. Wang, H. Wang, R. M. Kling, K. Ramakrishnan, and J. P. Shen, "Register renaming and scheduling for dynamic execution of predicated code," in *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, 2001.

[9] S. A. Mahlke, R. E. Hank, J. E. McCormick, D. I. August, and W.-M. W. Hwu, "A comparison of full and partial predicated execution support for ilp processors," in *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*. ACM, 1995.

[10] S. Friedman, A. Carroll, B. Van Essen, B. Ylvisaker, C. Ebeling, and S. Hauck, "SPR: an architecture-adaptive CGRA mapping tool," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2009.

[11] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "DRESC: a retargetable compiler for coarse-grained reconfigurable architectures," in *IEEE International Conference on Field-Programmable Technology*, 2002, pp. 166–173.

[12] D. Pellerin and S. Thibault, *Practical FPGA Programming in C*. Prentice Hall PTR, April 2005.

[13] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens, "Programmable Stream Processors," *IEEE Computer*, vol. 36, no. 8, pp. 54–62, 2003.

[14] B. Van Essen, A. Wood, A. Carroll, S. Friedman, R. Panda, B. Ylvisaker, C. Ebeling, and S. Hauck, "Static versus scheduled interconnect in Coarse-Grained Reconfigurable Arrays," in *International Conference on Field-Programmable Logic and Applications*, 31 2009-Sept. 2 2009, pp. 268–275.

[15] V. C. Sreedhar, G. R. Gao, and Y.-F. Lee, "Identifying loops using dj graphs," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 6, pp. 649–658, 1996.

[16] M. Smelyanskiy, S. A. Mahlke, E. S. Davidson, and H.-H. S. Lee, "Predicate-aware scheduling: a technique for reducing resource constraints," in *CGO '03: Proceedings of the international symposium on Code generation and optimization*. IEEE Computer Society, 2003.

[17] R. Cytron, J. Ferrante, and V. Sarkar, "Compact representations for control dependence," in *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. ACM, 1990.

[18] M. Budiu and S. C. Goldstein, "Compiling application-specific hardware," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2002.

[19] J. M. P. Cardoso, "Dynamic loop pipelining in data-driven architectures," in *CF '05: Proceedings of the 2nd conference on Computing frontiers*. ACM, 2005.