Merging Head and Tail Duplication for Convergent Hyperblock Formation

Bertrand A. Maher Aaron Smith Doug Burger Kathryn S. McKinley

Department of Computer Sciences
The University of Texas at Austin
{bmaher, asmith, dburger, mckinley}@cs.utexas.edu

Abstract

VLIW and EDGE (Explicit Data Graph Execution) architectures rely on compilers to form high-quality hyperblocks for good performance. These compilers typically perform hyperblock formation, loop unrolling, and scalar optimizations in a fixed order. This approach limits the compiler's ability to exploit or correct interactions among these phases. EDGE architectures exacerbate this problem by imposing structural *constraints on hyperblocks, such as instruction count and instruction composition.*

This paper presents convergent hyperblock formation, *which iteratively applies if-conversion, peeling, unrolling, and scalar optimizations until converging on hyperblocks that are as close as possible to the structural constraints. To perform peeling and unrolling, convergent hyperblock formation generalizes* tail duplication, *which removes side entrances to acyclic traces, to remove back edges into cyclic traces using* head duplication. *Simulation results for an EDGE architecture show that convergent hyperblock formation improves code quality over discrete-phase approaches with heuristics for VLIW and EDGE. This algorithm offers a solution to hyperblock phase ordering problems and can be configured to implement a wide range of policies.*

1. Introduction

Both VLIW [10] and EDGE architectures [6] rely on the compiler to produce hyperblocks—single-entry, multiple-exit sets of predicated instructions—that are well matched to the underlying architecture [18]. Compilers for these architectures perform transformations such as loop unrolling to expose parallelism among iterations and if-conversion [3] to expose parallelism among basic blocks [1, 10, 13, 16, 18, 24]. If-conversion replaces control dependences with data dependences, i.e., predicated instructions [3], to combine basic blocks into hyperblocks. The compiler further optimizes hyperblocks with scalar optimizations such as common subexpression elimination to reduce instruction counts.

Prior compilers selected a discrete ordering for loop unrolling, loop peeling, hyperblock formation, and scalar optimizations [1, 10, 13, 16, 18, 24]. However, if hyperblock formation occurs before loop unrolling, the compiler will not combine iterations to form larger hyperblocks. If it performs unrolling first, the compiler must predict the effects of hyperblock formation to pick an appropriate unroll factor. To form effective hyperblocks, the compiler must balance the interactions among these phases. Although August et al. introduce reverse if-conversion to resolve the phase ordering problem between hyperblock formation and scalar optimizations [5], but previous compiler research has not focused on the interaction of unrolling and peeling with hyperblock formation [1, 10, 18].

EDGE architectures complicate this phase ordering problem by placing additional *structural* constraints on hyperblocks. To simplify the hardware, hyperblocks have an architecturally-defined maximum number of instructions, maximum number of loads or stores, and restrictions on register accesses. The compiler seeks to fill each block as full as possible to amortize the runtime cost of mapping each fixed-size block to the hardware substrate. If an EDGE compiler conservatively forms hyperblocks that meet the constraints and then applies scalar optimizations that reduce code size, it misses opportunities to pack more instructions into the hyperblocks and better utilize the hardware instruction window.

Convergent hyperblock formation addresses the challenges of phase ordering and converging to structural constraints. This algorithm incrementally merges basic blocks and repeatedly applies scalar optimizations until it cannot add any block, thus converging on the limit of the structural constraints. To perform peeling and unrolling, convergent hyperblock formation generalizes *tail duplication* [7, 18]. Tail duplication eliminates a side entrance to an acyclic trace by duplicating code below a merge point. *Head duplication* eliminates the back edge of a loop (a side entrance

to a cyclic trace) by duplicating and predicating the target of the back edge. This process integrates peeling and unrolling with hyperblock formation.

The algorithm can implement a wide range of heuristics by carefully selecting the order to merge blocks. VLIW block selection heuristics have focused on minimizing and balancing dependence height, because dependences along any predicate path in a hyperblock constrain its static schedule height, even if that path does not execute at runtime. Because the EDGE microarchitecture supports dynamic instruction issue and allows a block to commit once it produces its outputs, the dependence height of a falsely predicated path has little effect on performance. EDGE heuristics instead perform best by creating full blocks, removing unpredictable branches, and limiting tail duplication.

We evaluate convergent hyperblock formation using simulation of the TRIPS EDGE architecture [6, 24]. These experiments show that this approach improves TRIPS microbenchmark cycle counts by 2 to 11% on average when compared to classical phase orderings. These results also establish a strong correlation between block count reduction and performance improvement. A functional simulator shows that convergent hyperblock formation reduces block counts of the SPEC2000 benchmarks, indicating potential performance improvement on real applications.

We measure a number of EDGE hyperblock policies and also measure VLIW heuristics implemented within the algorithm. Measuring VLIW and EDGE heuristics with and without phase ordering integration shows that integration improves the performance of both by resolving the phase ordering issues of hyperblock formation, loop peeling, loop unrolling, and scalar optimizations.

2. TRIPS architecture background

Explicit Dataflow Graph Execution (EDGE) architectures provide a hybrid dataflow execution model within a single thread of control that supports conventional imperative languages [6]. An EDGE program is compiled into a sequence of structured hyperblocks that each commit atomically. Within a block, the instructions explicitly encode their dependences in a static dataflow graph, using target form in source instructions rather than writing to shared registers. By mapping and executing multiple blocks at once, the microarchitecture forms large instruction windows.

The TRIPS processor architecture is one EDGE ISA that employs dynamic issue of instructions from all in-flight blocks. The TRIPS microarchitecture is a 16-wide processor with 128 architectural registers. Each block can contain up to 128 instructions, mapping eight instructions to each functional unit to minimize contention and communication latencies [8, 20]. Using speculative next-block prediction, the microarchitecture supports eight blocks in flight, making the maximum instruction window size 1024 instructions. The processor commits the oldest in-flight block after it produces all of its outputs: up to 32 stores, up to 32 register writes, and a single branch decision. Each block contains up to 32 register reads and writes in addition to the 128 regular instructions.

The TRIPS ISA places four restrictions on blocks intended to strike a balance between software and hardware complexity. They are: (1) a maximum block size of 128 instructions, (2) a maximum of 32 loads and stores may issue per block, (3) a maximum of eight reads and eight writes to each of four register banks per block, and (4) a per-block fixed number of block outputs (each block must always generate a constant number of register writes and stores, plus exactly one branch). The first three constraints are fixed by the block format, whereas the microarchitecture uses the fourth to detect block termination.

Previous work describes how the compiler uses SSA to guarantee that each block produces a fixed number of outputs [24]. The hyperblock formation algorithm used in that study forms hyperblocks conservatively, using a fixed phase ordering for hyperblock formation, peeling, unrolling, and scalar optimizations. Due to interactions among these phases, a conservative approach leaves many hyperblocks underfilled, thus motivating an alternative approach to fixed phase ordering.

3. Phase ordering challenges

Structural architectural constraints on hyperblock formation exacerbate a phase ordering problem between hyperblock formation, loop unrolling, loop peeling, and scalar optimization.

Hyperblock formation creates scalar optimization opportunities that are difficult to express in the control-flow domain. Two such examples are instruction merging, which combines instructions from distinct control-flow paths, and implicit predication, where the compiler predicates only the head instruction in a dependence chain, thus implicitly predicating the successors [25]. Since these optimizations typically eliminate instructions, their application may enable the compiler to include more basic blocks in the hyperblock, improving code density.

Loop unrolling and peeling present a similar challenge. If the compiler performs these transformations before hyperblock formation, it may if-convert multiple iterations and combine them into large hyperblocks. Without performing hyperblock formation on the body of the loop, however, the compiler cannot determine an appropriate unroll factor. Figure 1a shows a CFG consisting of an outer loop with two inner loops where all the loops must perform their exit test on each iteration. Such loops are termed *while loops*, but do not assume the C “while” loop construct. While-loop

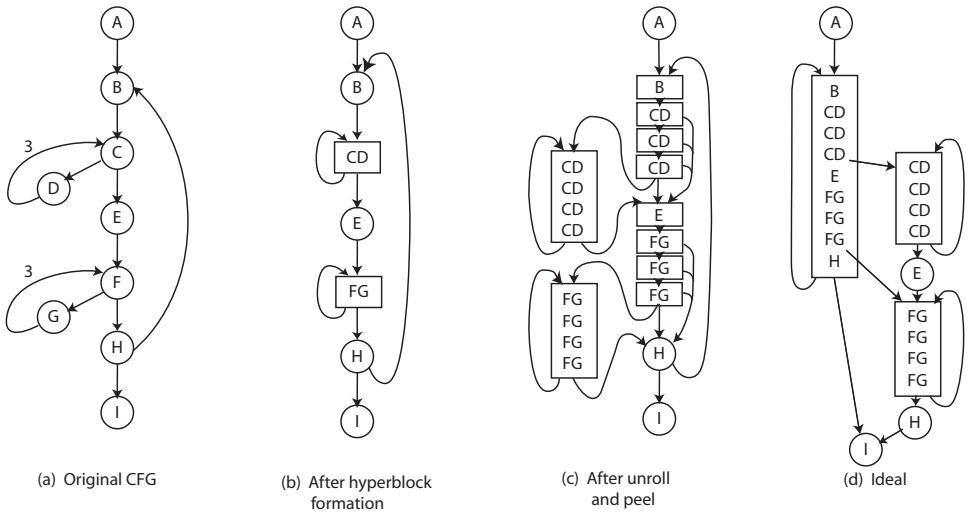

Figure 1. Hyperblock formation example.

unrolling requires hyperblock formation to predicate each iteration, unlike *for-loop* unrolling, which can remove intermediate tests. This example assumes that profiling indicates that each loop typically iterates three times.

Figure 1b shows hyperblock formation in which the compiler first if-converts the bodies of the inner while loops. Figure 1c shows the code when the compiler uses the profile to peel three iterations, and then unrolls the loop four times to fill the hyperblock for the less frequent case. Ideally, the compiler would now repeat hyperblock formation to produce the code in Figure 1d.

Under different conditions, the ideal phase ordering may include another pass of peeling: if the loops execute either three or four times in the common case and the compiler cannot fit four peeled iterations in a single block until after scalar optimizations, the ideal compiler would peel again. Thus, each static phase ordering of hyperblock formation, peeling, unrolling and scalar optimizations may miss opportunities to create better hyperblocks.

4. Convergent hyperblock formation

Convergent hyperblock formation iteratively and incrementally applies optimizations to ensure that each hyperblock is well-constructed and tightly packed with useful instructions. The algorithm incorporates loop peeling and unrolling by generalizing tail duplication to remove back edges.

4.1. Head and tail duplication

Compilers use tail duplication to expand a hyperblock by duplicating code below a merge point and eliminating side entrances. On a VLIW architecture, the compiler copies the merge point, and changes the pertinent branch to target the copy. Other than the branch, the compiler does not need to modify either the copied or the original code.

An EDGE compiler, however, must predicate the merge point for two reasons. First, a branch does not immediately terminate an EDGE hyperblock; instead, the architecture requires that a block produce all of its outputs (register writes and stores, in addition to the branch) before committing. Second, unpredicated instructions within the block execute when they receive operands. Therefore, the instructions in an unpredicated merge point would send results to the outputs of the block, even if the processor takes the side exit. Essentially, duplicating the merge point makes it control-dependent on the exit test, and correct dataflow execution requires the compiler to convert this control dependence to a data dependence.

Furthermore, the EDGE compiler must transform the resultant hyperblock to meet the structural constraints (i.e., the block must produce a fixed number of outputs) [24]. Thus the side exit must produce the same number of outputs as the main path, potentially adding size overhead to the hyperblock, and runtime overhead if the side exit is taken.

Figure 2a shows an example of tail duplication in which the compiler chooses to combine A, B, and D. The compiler first if-converts B and merges it with A (Figure 2b). The compiler then applies tail duplication to D to eliminate the side entrance. It copies D to create D' (Figure 2c). Next, it modifies the CFG, redirecting $AB \rightarrow D$ to D' (Figure 2d). The compiler then if-converts D', predicating the instructions on the original branch condition for $A \rightarrow B$, and merges the result into the hyperblock (Figure 2e).

Compilers typically perform hyperblock formation on acyclic regions within a CFG. Head duplication generalizes hyperblock formation to include cyclic regions, effectively

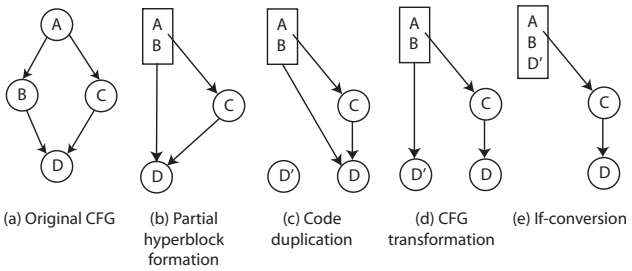

Figure 2. Classical tail duplication.

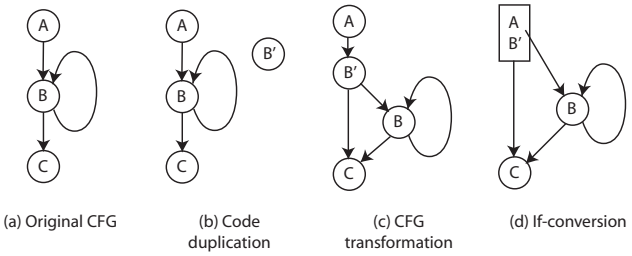

Figure 3. Head duplication implements peeling.

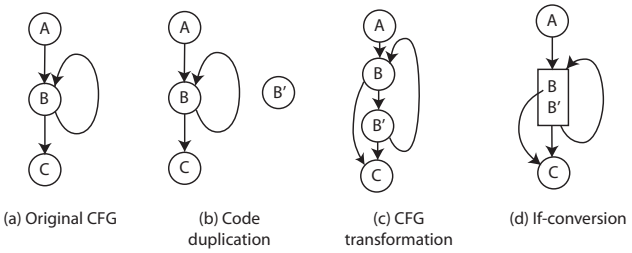

Figure 4. Head duplication implements unrolling.

implementing peeling and unrolling. Encountering a block that is the target of both an edge from the current hyperblock and a loop back edge is similar to encountering a block with a side entrance; thus the compiler can apply the same tail duplication process. Tail duplication creates outgoing edges from the duplicated block to the successors of the original. If the original block was a loop, the new edge will either be a loop entrance (peeling) or a back edge (unrolling).

Consider the CFG in Figure 3a. Since B is a loop header, tail duplication is insufficient for combining A and B. Head duplication peels a copy of B to merge with A. The compiler copies B to make B' (Figure 3b), then redirects edge $A \rightarrow B$ to B', adds $B' \rightarrow B$, and for all $B \rightarrow X$, inserts $B' \rightarrow X$ (Figure 3c). Finally, the compiler if-converts and merges B' and A (Figure 3d).

Figure 4 shows how head duplication also implements loop unrolling. Consider creating a hyperblock starting with basic block B. Since the back edge points to B, the compiler can unroll and still satisfy the single entry constraint. Head duplication creates a copy of the loop body B' (Figure 4b). The compiler then replaces $B \rightarrow B$ with $B' \rightarrow B$, inserts $B \rightarrow B'$, and $B' \rightarrow C$ (Figure 4c). The last step if-converts and merges B' into B (Figure 4d). If the compiler were to apply additional unrolling directly to this CFG, it could only unroll by powers of two. To remove this limitation, the unrolling procedure saves the original loop body and appends one additional iteration at a time.

4.2. Incremental hyperblock formation

Convergent hyperblock formation integrates unrolling, peeling, tail duplication, if-conversion and scalar optimizations to form hyperblocks incrementally. Figure 5 shows the pseudocode for the algorithm. The procedure `ExpandBlock` starts with a block HB and selects a successor S to merge according to a heuristic. The compiler then calls `MergeBlocks`, which attempts to merge S into HB, duplicating code if necessary. If the merge is successful, the compiler adds the successors of S to the set of candidates.

`MergeBlocks` first copies HB and S to scratch space and attempts to if-convert and merge S into HBS. The compiler optimizes the resulting block, and then checks to determine whether the block violates the structural constraints. If so, the merge fails and the compiler considers other successors. By testing the merge in scratch space before transforming the CFG, the implementation avoids a more complicated undo step.

If the merge is successful, the compiler must transform the CFG appropriately. If $HB \rightarrow S$ is the only entrance to S, the compiler can simply remove S from the CFG and replace HB with HBS (lines 7–9 in `MergeBlocks`). Otherwise, it must perform code duplication. Lines 10–15 of `MergeBlocks` show the cases where the compiler performs unrolling, peeling, and tail duplication. The compiler uses head duplication to implement unrolling and peeling and tail duplication for other cases. The `Optimize` step attempts to eliminate instructions in the merged block. The compiler currently applies dominator-based global value numbering and predicate optimizations that reduce the number of instructions that use each predicate [25].

5. Policy

Convergent hyperblock formation constructs hyperblocks that obey architectural constraints while maximizing code density. To achieve high performance, however, the algorithm must apply heuristics that select the most profitable basic blocks to include in each hyperblock. This block selection policy can balance several characteristics of high-performance hyperblocks.

The two simplest heuristics, breadth-first and depth-first, each emphasize one of two opposing goals. By merging basic blocks in breadth-first order, the compiler guarantees the inclusion of some useless instructions, but attempts to decrease the branch misprediction frequency and limit tail

```
procedure ExpandBlock(HB : block)
 1: candidates := Successors(HB)
 2: while candidates is not empty do
 3:    S := SelectBest(candidates)
 4:    candidates := candidates − {S}
 5:    if not LegalMerge(HB, S) then
 6:       continue
 7:    else if MergeBlocks(HB, S) == Success then
 8:       candidates := candidates ⋃ Successors(S)
 9:    end if
10: end while

procedure MergeBlocks(HB, S : block)
 1: HBcopy := Copy(HB)
 2: Scopy := Copy(S)
 3: HBS := Combine(HBcopy, Scopy)
 4: Optimize(HBS)
 5: if not LegalBlock(HBS) then
 6:    return Failure
 7: else if NumPredecessors(S) == 1 then
 8:    Replace(HB, HBS) // no code duplication
 9:    Remove(S)
10: else if HB → S is a back edge and HB == S then
11:    UnrollLoop(HB, S)
12: else if S is a loop header and HB → S is not a back
    edge then
13:    PeelLoop(HB, S)
14: else
15:    TailDuplicate(HB, S)
16: end if
17: return Success
```

Figure 5. Convergent hyperblock formation algorithm.

duplication. The depth-first policy risks a higher misprediction rate and performs more tail duplication, but seeks to include a greater number of useful instructions.

Branch predictability: Removing conditional branches is important for EDGE architectures because of their large instruction windows. In the TRIPS prototype, each processor has a 1024-instruction window consisting of eight blocks, seven of which are speculative. A branch misprediction and subsequent pipeline flush prevents effective utilization of this window. The compiler can improve predictability during hyperblock formation by eliminating unpredictable conditional branches. One heuristic that eliminates conditional branches is to end blocks at merge points so that each has a single exit, however this policy may result in under-full hyperblocks.

Limiting tail duplication: On a dataflow architecture, tail duplication requires additional predication below the side exit, including predication of the merge point. This requirement introduces data dependences on the outcome of the test in the duplicated code, while in the original program the instructions were control independent. These dependences may degrade performance, since the resultant code cannot execute speculatively, but must wait on the resolution of a possibly time-consuming test. This effect is especially problematic when the duplicated merge point contains a loop induction variable update that is on the critical path through an otherwise parallel loop.

Loop peeling and unrolling: Because convergent hyperblock formation folds loop transformations into the hyperblock formation algorithm, the compiler can apply block selection policies to loops as well. To perform peeling accurately, the compiler can use loop trip count histograms to augment an edge frequency profile. A loop peeling policy can then evaluate the benefit of unrolling additional loop iterations versus including post-loop code by using a threshold function to pick an appropriate peeling factor.

Local and global heuristics: Local heuristics consider only the characteristics of the current hyperblock when choosing among the candidate successors. Because of the architectural constraints on TRIPS hyperblocks, a local approach works well for TRIPS block formation. By incrementally merging basic blocks, the hyperblock gradually converges on the upper bound of the constraints. Because the compiler adds blocks individually to satisfy structural constraints, the algorithm focuses on selecting one of a block's immediate successors for inclusion.

Using lookahead can increase the power of local heuristics. For example, a heuristic that improves branch predictability favors blocks with a single exit. Such a heuristic might first determine if a hyperblock has one exit, and then use lookahead to estimate if the compiler can include enough additional basic blocks to reach the next merge point, thus constructing a larger, single-exit hyperblock.

Although local heuristics seem most suitable for incremental hyperblock formation, the algorithm can use global information to inform block selection by performing a prepass analysis. To implement path-based VLIW heuristics [17, 18] using convergent hyperblock formation, the compiler analyzes the CFG to create a prioritized list of basic blocks, and then merges blocks in priority order, when possible.

Dependence height: The best-known block selection heuristic for VLIW architectures analyzes all paths through a region to determine which basic blocks to include [17]. Because a VLIW hyperblock is statically scheduled, the dependence height of the longest path determines the execution time of the block, even if that path is not taken at runtime. Paths are therefore prioritized to favor those that execute frequently, consume few resources, and have short dependence heights. These heuristics attempt to avoid over-constraining the static schedule or over-saturating the

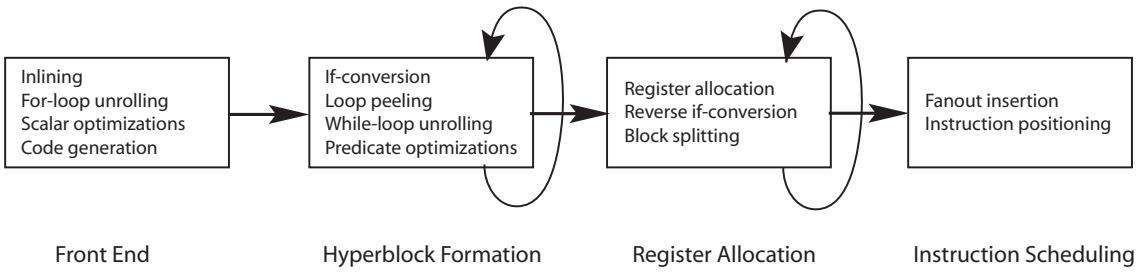

Figure 6. Compiler flow with convergent hyperblock formation.

processor's resources, while still including the most useful paths and removing unpredictable branches.

For EDGE ISAs, minimizing dependence height is less important. EDGE instructions issue dynamically when their operands arrive, and the architecture can commit a block as soon as it produces its outputs. Therefore, if a short path through an EDGE hyperblock completes before a longer one, the architecture detects block termination and does not wait for the longer path to finish. Although speculative instructions on an untaken path may contend for resources with instructions on a taken path, this contention reflects constraints on issue width rather than on schedule height.

6. TRIPS compiler

We implement convergent hyperblock formation in Scale [19, 24], a retargetable compiler with a back end for TRIPS. Figure 6 shows the overall compiler flow. The compiler front end operates on a language and machine-independent control-flow graph representation. Scale performs inlining and for-loop unrolling first, followed by classical scalar optimizations. The compiler then lowers the program representation to a RISC-like form. Using this representation, the compiler performs hyperblock formation, followed by register allocation, fanout insertion to replicate values for multiple consumers, and finally scheduling.

Since register allocation and fanout insertion add additional instructions and occur after hyperblock formation, the compiler must estimate final block sizes while forming hyperblocks. Although hyperblock formation tries to construct blocks of the appropriate size and load/store count, the register allocator may insert spill code that violates the block constraints. If a block has spills that cause it to violate a constraint, the compiler performs reverse if-conversion on the block, and repeats register allocation. Scale rarely needs to split blocks in this manner, both because TRIPS has a large number of architectural registers and because the compiler attempts to avoid inserting spill code in nearly full hyperblocks. Once register allocation completes and all blocks are valid, the scheduler inserts fanout instructions, assigns locations to all instructions in their respective blocks, and translates them to TRIPS assembly language.

7. Experimental results

We evaluate convergent hyperblock formation using a TRIPS cycle-level timing simulator, which has been verified to be within 4% of the cycle counts generated by the TRIPS prototype hardware design on a set of microbenchmarks. This simulator models all aspects of the microarchitecture, including global control, data path pipelines, and communication delays within the processor. Because detailed simulation is prohibitively slow (approximately 1000 instructions per second), we restrict the evaluation to microbenchmarks derived by extracting loops and procedures from SPEC2000, and with signal-processing kernels from the GMTI radar suite, a 10x10 matrix multiply, sieve (a prime number generator), and Dhrystone.

7.1. Comparison to static phase ordering

Table 1 compares the performance of hyperblock formation with discrete phases of unrolling and if-conversion to the single-phase, but iterative, convergent hyperblock formation. Column 2 shows the baseline cycle count of each benchmark using basic blocks as TRIPS blocks. Basic blocks are a good baseline because they are defined by the front end of the compiler, and do not depend on the hyperblock formation algorithm or heuristics. The remaining columns apply if-conversion (I), unrolling/peeling (UP), and scalar optimizations (O) in various orders. Phases grouped together in parentheses indicate that the transformations are applied incrementally, using head duplication to implement unrolling and peeling, and iterative optimization to improve code density. All results use a greedy breadth-first policy and use incremental if-conversion to avoid violating the block constraints.

The UPIO and IUPO columns show discrete phase orderings of structural transformations followed by scalar optimizations. UPIO performs loop unrolling and peeling before incremental if-conversion and tail duplication. This approach improves performance by an average of 16% over basic blocks. The second phase ordering (IUPO in Column 4) performs if-conversion before loop unrolling and peeling. It improves performance an additional 8.8% on aver-

	BB	UPIO		IUPO		(IUP)O		(IUPO)	
	cycles	m/t/u/p	%	m/t/u/p	%	m/t/u/p	%	m/t/u/p	%
ammp_1	1544356	18/11/3/0	18.2	18/11/11/7	68.6	18/11/11/7	68.6	18/11/13/8	65.9
ammp_2	1021042	39/11/3/0	13.6	39/11/8/3	59.0	39/11/8/3	59.0	40/2/10/2	60.2
art_1	83309	11/1/3/5	4.5	12/0/3/4	12.1	13/0/3/2	4.9	13/0/4/2	6.0
art_2	128499	4/1/2/5	-7.5	6/0/2/2	1.0	8/1/2/1	-5.3	7/1/4/3	3.4
art_3	638918	23/1/2/3	76.6	24/0/3/2	77.0	25/1/3/1	74.4	25/0/3/2	76.1
bzip2_1	478746	7/0/0/0	22.1	7/0/0/0	22.1	7/0/0/0	22.1	7/0/0/0	22.1
bzip2_2	334299	9/1/3/5	32.6	10/0/3/5	32.0	10/0/3/5	32.0	10/1/3/5	32.0
bzip2_3	556743	10/0/3/0	34.6	10/0/3/1	34.6	10/0/3/1	34.6	10/0/3/1	34.5
dct8x8	51988	4/0/0/0	-0.6	4/0/0/0	-0.6	4/0/0/0	-6.3	4/0/0/0	-6.3
dhry	234345	63/2/4/5	13.5	64/3/6/9	22.5	66/4/6/8	23.1	64/4/10/12	23.5
doppler_GMTI	85229	7/2/10/9	21.8	8/1/11/9	13.2	8/1/11/9	14.3	8/1/12/11	16.6
equake_1	114324	6/0/0/0	0.7	6/0/0/0	0.7	6/0/0/0	2.3	7/0/0/0	12.4
fft2_GMTI	130496	11/3/1/3	25.9	12/2/1/2	21.1	13/3/1/1	25.2	13/4/1/1	27.9
fft4_GMTI	98538	7/1/0/1	4.7	8/0/0/0	7.3	8/0/0/0	6.6	8/1/0/0	4.9
forward_GMTI	180900	10/2/1/3	0.5	10/3/1/4	2.2	11/4/1/3	2.0	11/3/3/7	3.8
gzip_1	29377	9/3/0/0	22.2	9/3/0/0	22.2	9/3/0/0	20.8	12/2/0/0	48.4
gzip_2	98414	5/2/3/6	54.8	6/2/3/4	46.4	7/2/3/2	48.3	6/2/6/7	54.9
matrix_1	71814	10/0/0/1	-25.2	10/0/3/5	37.9	10/1/3/5	38.4	10/1/4/6	42.3
parser_1	395076	12/0/0/0	46.5	12/0/0/0	46.5	12/0/0/0	46.5	12/0/0/0	46.5
sieve	443064	6/3/7/8	-13.1	7/1/7/4	20.9	7/2/7/4	23.7	7/2/9/5	22.6
transpose_GMTI	185803	6/0/0/0	4.2	6/0/0/0	4.2	6/0/0/0	1.6	7/1/0/1	1.5
twolf_1	527166	10/5/1/2	38.9	11/4/1/1	39.7	12/4/1/0	38.9	15/1/1/0	38.6
twolf_3	588011	12/0/0/0	0.5	12/0/0/0	0.5	12/0/0/0	0.5	12/0/0/0	0.5
vadd	105407	5/1/0/1	-2.1	5/1/1/5	7.9	6/2/1/5	5.4	6/2/2/5	9.4
Average			**16.2**		**25.0**		**24.2**		**27.0**

Table 1. Percent improvement in cycle counts of hyperblocks over basic blocks (BB) and static count of blocks merged/tail duplicated blocks/unrolled iterations/peeled iterations (m/t/u/p), with various orderings of Unrolling (U), Peeling (P) Incremental If-conversion (I), and Scalar Optimizations (O). Parentheses indicate merged phases.

age compared to UPIO since the unroller has more accurate block counts and size estimates for loops with control flow after if-conversion than before.

The (IUP)O column shows that iterating peeling and unrolling appears to offer no benefit over the distinct phases in IUPO on these benchmarks, despite adding the capability to generate hyperblocks like Figure 1d. Most of these benchmarks consist simply of for loops with high trip counts. Because Scale applies for-loop unrolling in the front end, the only benefit of head duplication is to merge the test for the execution of the post-conditioning loop with the body of the unrolled loop. Sometimes merging this test helps performance slightly (e.g., fft2_GMTI and sieve), and sometimes it hurts slightly (e.g., art_1 and art_2). The best candidates for head duplication are ammp_1 and ammp_2, which contain while loops with low trip counts. However, the compiler's block size estimates are not yet sufficiently accurate to combine peeled iterations of these loops with surrounding code (see Section 6).

Integrating scalar optimizations into hyperblock formation (the (IUPO) column) attains an additional 2% performance improvement because the compiler can pack blocks more tightly and perform more if-conversion and unrolling. The most significant improvement, gzip_1, occurs because the compiler uses if-conversion and scalar optimizations to fit the entire body of the innermost loop in one block, dramatically reducing the total number of blocks executed.

The m/t/u/p statistics show how often the compiler applies if-conversion, tail duplication, unrolling, and peeling. For example, the performance improvement of ammp_1 in all columns following UPIO occurs because the compiler unrolls and peels several additional iterations of the critical loops. Using (IUPO) on ammp_1 enables peeling of an additional loop iteration, but this transformation happens to create a less-predictable branch pattern and increases the number of mispredicted branches by 50%.

On the microbenchmarks, the best heuristic for forming TRIPS hyperblocks improves performance compared to basic blocks by 27% on average. Convergent hyperblock formation outperforms classical optimization phase orderings by an average of between 2 and 11%.

7.2. VLIW and EDGE heuristics

The above results show that convergent hyperblock formation offers potential performance benefits given the right heuristics. Convergent hyperblock formation is flexible enough to implement a variety of policies as discussed in Section 5. We compare performance using three heuristics: the VLIW heuristic proposed by Mahlke et al. [17, 18], a

	BB	VLIW	Convergent VLIW	DF	BF
ammp_1	1544356	64.8	61.7	62.8	65.9
ammp_2	1021042	3.8	4.1	1.7	60.2
art_1	83309	3.3	2.6	7.0	6.0
art_2	128499	0.3	7.2	6.9	3.4
art_3	638918	45.0	45.0	29.3	76.1
bzip2_1	478746	-25.4	-25.4	-37.4	22.1
bzip2_2	334299	-59.0	0.9	-40.6	32.0
bzip2_3	556743	-67.9	-68.1	-91.7	34.5
dct8x8	51988	-0.6	18.3	-7.5	-6.3
dhry	234345	17.2	17.2	19.6	23.5
doppler_GMTI	85229	13.1	16.6	19.7	16.6
equake_1	114324	0.7	13.6	12.4	12.4
fft2_GMTI	130496	28.0	28.0	28.7	27.9
fft4_GMTI	98538	5.6	6.6	10.2	4.9
forward_GMTI	180900	4.7	-1.0	5.4	3.8
gzip_1	29377	49.3	46.1	12.1	48.4
gzip_2	98414	30.1	29.2	32.0	54.9
matrix_1	71814	37.9	39.2	40.0	42.3
parser_1	395076	25.1	27.0	45.1	46.5
sieve	443064	11.2	16.6	1.5	22.6
transpose_GMTI	185803	4.2	-0.9	2.5	1.5
twolf_1	527166	-60.8	-42.6	-41.9	38.6
twolf_3	588011	7.4	3.7	4.8	0.5
vadd	105407	7.9	11.2	15.1	9.4
Average		**6.1**	**10.7**	**5.7**	**27.0**

Table 2. Percent improvement in cycle count over basic blocks (BB) using VLIW heuristics, VLIW with iterative optimization, depth-first (DF) and breadth-first (BF) EDGE heuristics.

depth-first heuristic that selects the most frequent path, and a breadth-first heuristic that removes conditional branches.

Table 2 shows the performance of the VLIW and EDGE heuristics on TRIPS. Columns 3 and 4 show the VLIW block selection heuristic applied without and with iterative optimization, respectively. Without iterative optimization the VLIW heuristic achieves a 6.1% average speedup over basic blocks, compared to 10.7% with iterative optimization, demonstrating that convergent hyperblock formation improves the performance of this heuristic. Column 5 shows the depth-first heuristic, which achieves a small 5.7% speedup. Breadth-first merging shows the greatest improvement, at 27%.

Several of the largest performance differences among these results occur because tail duplication incurs additional predication on a dataflow architecture. The most extreme example of this effect is bzip2_3, where breadth-first merging achieves a 34.5% speedup while depth-first and VLIW degrade performance by 68.1% and 91.7%, respectively. While breadth-first merges all paths through the main loop, the depth-first and VLIW heuristics exclude an infrequently-taken block, and therefore must tail duplicate the final block in the loop, which contains the induction variable increment. The induction variable is then data-dependent on the earlier test, instead of being independent. This dependence results in a slowdown even over basic blocks, where the increment can be executed speculatively.

Improved branch prediction accuracy is another important effect. In parser_1, the VLIW heuristic excludes several rarely taken paths with relatively large dependence heights. Because these branches are rarely taken, they cause mispredictions when they occur, resulting in an 11-fold increase in the misprediction rate (0.4% using breadth-first versus 4.5% with VLIW), which reduces the effective size of the processor's issue window. The depth-first heuristic does not suffer branch mispredictions because it is able to include all paths through the loop, since there is ample space in the block after merging the most frequent path.

7.3. Estimated performance with block counts

The cycle-level simulator is too slow to simulate the full SPEC benchmarks. Since successful transformations reduce the number of blocks executed, thus increasing the issue window utilization and decreasing block overhead, block counts and program cycle counts should correlate. We demonstrate this correlation and present block count results for SPEC2000.

The best static phase ordering achieves a 2.1x improvement in number of blocks executed over basic blocks on the microbenchmarks, while iterative hyperblock formation achieves a 2.3x improvement. To first order, the relationship between the number of blocks executed and the cycle count is roughly: $cycles_{total} = cycles_{base} + blocks \times overhead$,

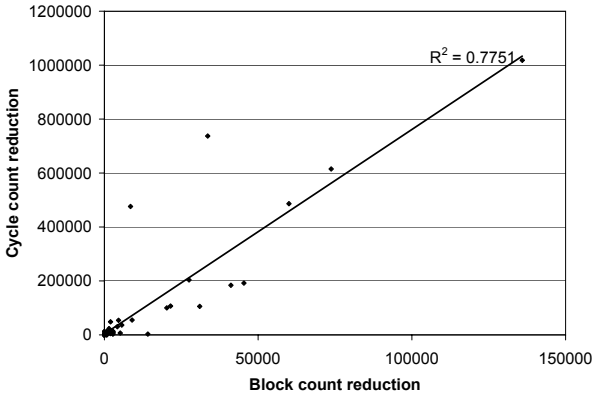

Figure 7. Cycle count reductions versus block count reductions.

where $cycles_{total}$ is the total number of cycles the program takes to execute, $cycles_{base}$ is the number of cycles to perform the computation (ignoring block boundaries), $blocks$ is the number of blocks, and $overhead$ is the fixed architectural overhead associated with mapping a block.

This equation is an oversimplification, because it does not account for increased parallelism exposed by block merging, nor interference from including speculative, useless instructions in a block. To evaluate the accuracy of this estimate, Figure 7 plots the change in cycle counts against the change in block counts (as compared to basic blocks) for all the data presented in Table 1. The relationship between block count reduction and cycle count reduction is roughly linear ($r^2 = 0.78$ using a linear regression test), with a few outliers due to ammp_1, in which reducing the block count by unrolling while loops dramatically improves performance. This result suggests that reduction in block count is a good but imperfect metric of performance improvement.

The correlation between block count reduction and cycle count reduction justifies the use of block counts (gathered using a fast, functional simulator) to estimate the performance effect of these algorithms on the SPEC2000 benchmarks. Table 3 shows the block count results for 19 of 21 FORTRAN and C benchmarks (at the time of this writing, the toolchain is not stable for 176.gcc and 253.perlbmk), where the baseline (BB) measures millions of basic blocks executed. The remaining columns report percent improvement, using the same configurations and format as Table 1. These results use the MinneSPEC small reduced dataset [14], since the ref datasets require too much simulation time, even using a less detailed simulator. The block count results show the same trends as the cycle-accurate results on the microbenchmarks, although head duplication is in general more effective and scalar optimizations are slightly less effective than in the microbenchmark results.

8. Related work

The closest related research is on algorithms for creating blocks to use as compilation units, and approaches for dealing with phase interactions.

8.1. Block formation algorithms

Fisher pioneered *trace scheduling* for VLIW architectures [10]. This compiler uses static branch prediction to identify frequently executed program paths (traces) at compile-time. It optimizes traces by removing constraints and pushing correcting code on to the less frequently taken paths. While effective, its drawbacks include code and compiler complexity due to side trace entrances [13] and the possibility of exponential code expansion [15].

Superblock scheduling [13] improves over trace scheduling by eliminating the need for side entrances by using tail duplication, which replicates instructions below a merge point, and redirects the side entrance to this copy. Treegions [12] are also single-entry, multiple-exit regions, but are larger than superblocks because they include basic blocks from many paths of control, thus limiting the effects of variations in execution profiles. The speculative hedge heuristic for superblock scheduling similarly attempts to minimize execution time across all paths to account for such variations [9].

The hyperblock [18] generalizes the superblock to enable effective use of predicated execution. The goal of hyperblock formation is to eliminate branching and maximize ILP, while avoiding over-committing processor resources [17]. Mahlke et al. perform heuristic-driven hyperblock formation, followed by block-enlargement optimizations (such as predicated loop peeling and hyperblock loop unrolling), and finally apply dataflow optimizations modified to operate on predicated code.

VLIW heuristics [5, 18, 22] focus on balancing dependence height, dependence width (resource utilization of each VLIW instruction and other resources), path frequency, and branch predictability. The hyperblock formation goals for VLIW and EDGE architectures are related since both attempt to apply if-conversion to expose scheduling (or placement) regions by removing branches. Path frequency is important to both architectures, since it is better to fill a hyperblock with instructions that execute under the same conditions and frequencies.

There are two important differences between VLIW and EDGE constraints. First, TRIPS blocks also must conform to the TRIPS architectural restrictions on block size, load/store ids, and register usage. Second, since VLIW machines issue instructions in a statically determined order, there is a high penalty for imbalanced dependence chains. An EDGE hyperblock can commit when all of its out-

		Phased		Convergent	
	BB (M)	UPIO	IUPO	(IUP)O	(IUPO)
ammp	12.5	65.4	72.9	73.9	73.9
applu	1.6	42.4	42.4	43.9	45.8
apsi	5.9	47.3	47.3	47.5	48.9
art	331.8	65.0	65.0	70.1	72.6
bzip2	248.8	40.8	45.9	46.5	50.4
crafty	16.7	42.7	46.5	49.3	55.3
equake	100.1	57.3	57.8	58.1	59.0
gap	20.6	11.7	11.8	11.8	11.9
gzip	86.5	59.4	60.0	60.0	62.3
mcf	28.1	61.2	69.8	70.1	63.8
mesa	870.3	51.8	51.8	51.8	51.8
mgrid	591.6	4.3	4.3	4.5	5.3
parser	87.2	51.7	57.0	57.1	57.1
sixtrack	479.3	54.1	54.2	54.2	53.5
swim	2.8	30.1	30.1	30.1	31.7
twolf	15.6	57.4	59.6	60.6	62.1
vortex	41.2	63.7	63.8	63.8	62.9
vpr	2.9	60.1	61.2	61.5	62.8
wupwise	1469.7	47.5	46.9	49.2	52.9
Average		**48.1**	**49.9**	**50.7**	**51.8**

Table 3. Percent improvement in block counts of SPEC benchmarks over basic blocks (BB) with various combinations and orderings of Unrolling (U), Peeling (P) If-conversion (I), and Scalar Optimizations (O). Parentheses indicate merged phases.

puts are produced, and thus long dependence chains with a false predicate do not directly add to the execution schedule length, although they do occupy space in a hyperblock that may otherwise have been filled with useful instructions.

Block-structured ISAs [11] also implement a block atomic execution model to exploit speculation and support wide issue. Instructions are not predicated but execute speculatively, with the processor aborting mispredicted blocks. The compiler attempts to combine basic blocks to increase the processor's fetch bandwidth and scheduling abilities, using an incremental technique that combines basic blocks until meeting a threshold: block size and number of exits. These resource constraints are more similar to a single VLIW instruction than a structured EDGE hyperblock. The block enlargement phase is similar to the incremental block merging in convergent formation, but the major phases differ because the block-structured ISA compiler does not include if-conversion, loop unrolling/peeling, or scalar optimizations in its iterative loop.

8.2. Solutions to phase ordering problems

August et al. discuss the interaction of hyperblock formation and scalar optimizations [5]. Their solution iterates on if-conversion, scalar optimizations, and VLIW scheduling. If this algorithm produces a poor schedule, it performs reverse if-conversion to remove basic blocks that constrain the schedule, allowing the algorithm to adjust hyperblock formation decisions after scheduling. Convergent hyperblock formation, by contrast, makes decisions incrementally in a single pass to ensure that the block conforms to the architectural constraints.

Another related technique is software pipelining [1, 2, 16, 23], which uses an iterative approach to select an appropriate unroll factor based on scheduling constraints. Rau's iterative modulo scheduling algorithm performs software pipelining for progressively larger values of the iteration interval until it resolves dependence constraints. Convergent hyperblock formation unrolls incrementally to fill blocks, but does not consider inter-iteration dependences.

9. Conclusions

While convergent hyperblock formation provides a method for building optimized hyperblocks, additional research should investigate block selection policies that better weigh block frequency, branch predictability, dependences created by tail duplication, and loop trip counts. Additional mechanisms (basic block splitting, for-loop unrolling, reentrant hyperblocks, and partial inlining) have the potential to increase the ability of convergent hyperblock formation to build effective and full structured hyperblocks. An open question is how to integrate these mechanisms into a single algorithm that does not have phase ordering issues.

Basic block splitting: Most hyperblock formation algorithms include basic blocks in a hyperblock in their entirety. However, because of the TRIPS structural constraints, the compiler may not be able to fit an entire basic block into

the current hyperblock. To solve this problem, the compiler could split large basic blocks and merge each part with a different hyperblock to improve code density. The policy that guides a block-splitting mechanism should consider carefully where to split the basic block. Within a block, temporary values do not consume architectural registers due to direct instruction communication, but values communicated between blocks must be stored in registers or memory, consuming resources. When splitting blocks, the compiler should seek to minimize cross-block communication, thus minimizing register pressure and the resultant spills.

For-loop unrolling: While-loop unrolling, which convergent hyperblock formation incorporates, requires the compiler to predicate each loop iteration. The Scale compiler also performs for-loop unrolling early in compilation to select an appropriate unroll factor using data from previous compilations [21]. Implementing incremental for-loop unrolling with hyperblock formation in the back end will more accurately target the block constraints and consequently produce better hyperblocks.

Re-entrant hyperblocks: The classical definition of a hyperblock does not permit side entrances. However, the compiler may introduce conventions for passing predicate information across blocks that correctly handle side entrances, violating the single-entry constraint to reduce code expansion [4]. For instance, a hyperblock at the head of a loop could perform different actions depending on whether it is entered from outside the loop or via the back edge.

Partial inlining: Compilers typically inline a function entirely or do not inline it at all. While the complexity of partial inlining—merging some basic blocks of the callee into the caller—discourages its implementation in compilers for an architecture with low function call overhead, structurally-constrained ISAs may require partial inlining to fill blocks aggressively.

Convergent hyperblock formation addresses the phase ordering problems created by generating blocks that are well-matched to the underlying architecture. By integrating loop unrolling and peeling using head duplication, and by performing scalar optimizations incrementally during basic block merging, the compiler can construct more densely packed hyperblocks that obey structural constraints imposed by the ISA. As technology scaling moves architecture towards communication-dominated designs requiring such ISAs, the compiler's role in effectively mapping code structures to the underlying microarchitecture will only become more important.

Acknowledgments

This research was supported financially by the Defense Advanced Research Projects Agency under contracts F33615-01-C-1892 and NBCH30390004, NSF instrumentation grant EIA-9985991, NSF CAREER grants CCR-9985109 and CCR-9984336, IBM University Partnership awards, grants from the Alfred P. Sloan Foundation and the Intel Research Council.

References

[1] A. Aiken and A. Nicolau. A realistic resource-constrained software pipelining algorithm. In *Workshop on Languages and Compilers for Parallel Computing*, Irvine, CA, Aug. 1990.

[2] A. Aiken, A. Nicolau, and S. Novack. Resource-constrained software pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 6(12):1248–1270, 1995.

[3] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *ACM Symposium on the Principles of Programming Languages*, Austin, TX, Jan. 1983.

[4] D. I. August. Hyperblock performance optimizations for ILP processors. Master's thesis, University of Illinois at Urbana-Champaign, 1993.

[5] D. I. August, W.-M. W. Hwu, and S. A. Mahlke. The partial reverse if-conversion framework for balancing control flow and predication. *International Journal of Parallel Programming*, 27(5):381–423, 1999.

[6] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, 2004.

[7] P. P. Chang, S. A. Mahlke, and W.-M. W. Hwu. Using profile information to assist classic code optimizations. *Software Practice and Experience*, 21(12):1301–1321, 1991.

[8] K. Coons, X. Chen, S. Kushwaha, D. Burger, and K. S. McKinley. A spatial path scheduling algorithm for EDGE architectures. In *International Conference on Architectural Support for Programming Languages and Operation Systems*, San Jose, CA, Oct. 2006.

[9] B. L. Deitrich and W.-M. W. Hwu. Speculative hedge: regulating compile-time speculation against profile variations. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 70–79, 1996.

[10] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.

[11] E. Hao, P.-Y. Chang, M. Evers, and Y. N. Patt. Increasing the instruction fetch rate via block-structured instruction set architectures. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 191–200, 1996.

[12] W. Havanki, S. Banerjia, and T. Conte. Treegion scheduling for wide issue processors. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 266, 1998.

[13] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery.

The superblock: an effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(1-2):229–248, 1993.

[14] A. KleinOsowski and D. Lilja. Minnespec: A new spec benchmark workload for simulation-based computer architecture research, 2002.

[15] J. Lah and D. E. Atkins. Tree compaction of microprograms. *SIGMICRO Newsletter*, 14(4):23–33, 1983.

[16] M. Lam. Software pipelining: An effective scheduing technique for VLIW machines. In *ACM Conference on Programming Language Design and Implementation*, Atlanta, GA, June 1988.

[17] S. A. Mahlke. *Exploiting instruction level parallelism in the presence of conditional branches*. PhD thesis, University of Illinois at Urbana-Champaign, 1997.

[18] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, 1992.

[19] K. S. McKinley, J. Burrill, M. Bond, D. Burger, B. Maher, B. Robatmili, and A. Smith. The Scale compiler, 2007. http://ali-www.cs.umass.edu/∼scale/.

[20] R. Nagarajan, D. Burger, K. S. McKinley, C. Lin, S. W. Keckler, and S. K. Kushwaha. Static placement, dynamic issue (SPDI) scheduling for EDGE architectures. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 74–84, Antibes Juan-les-Pins, France, Oct. 2004.

[21] N. Nethercote, D. Burger, and K. S. McKinley. Convergent compilation applied to loop unrolling. *HiPEAC Journal*. To appear.

[22] J. C. H. Park and M. Schlansker. On predicated execution. Technical Report HPL-91-58, HP Laboratories, 1991.

[23] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *Proceedings of the 27th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 63–74, 1994.

[24] A. Smith, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, K. S. McKinley, and J. Burrill. Compiling for EDGE architectures. In *Fourth International IEEE/ACM Symposium on Code Generation and Optimization*, pages 185–195, 2006.

[25] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. Dataflow predication. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2006.