Exploiting Postdominance for Speculative Parallelization

Mayank Agarwal, Kshitiz Malik, Kevin M. Woley, Sam S. Stone and Matthew I. Frank
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
{magarwa2,kmalik1,woley,ssstone2,mif}@uiuc.edu

Abstract

Task-selection policies are critical to the performance of any architecture that uses speculation to extract parallel tasks from a sequential thread. This paper demonstrates that the immediate postdominators *of conditional branches provide a larger set of parallel tasks than existing task-selection heuristics, which are limited to programming language constructs (such as loops or procedure calls). Our evaluation shows that postdominance-based task selection achieves, on average, more than double the speedup of the best individual heuristic, and 33% more speedup than the best combination of heuristics.*

The specific contributions of this paper include, first, a description of task selection based on immediate postdominance for a system that speculatively creates tasks. Second, our experimental evaluation demonstrates that existing task-selection heuristics based on loops, procedure calls, and if-else statements are all subsumed by compiler-generated immediate postdominators. Finally, by demonstrating that dynamic reconvergence prediction closely approximates immediate postdominator analysis, we show that the notion of immediate postdominators may also be useful in constructing dynamic task selection mechanisms.

1 Introduction

Speculative parallelization enables the effective automatic parallelization of programs that are hard to parallelize statically, usually because the compiler cannot conservatively prove that the resulting tasks would be completely data independent. The success of such systems depends, to a large extent, on the policies used to extract parallel tasks from the program. Most speculative parallelization systems use heuristics that exploit program structure to break the program execution into concurrent tasks. Common heuristics include creating (*spawning*) new tasks from individual loop iterations, from the fall-throughs of loops and procedures, or from some combination of these techniques.

Intuitively, these task selection policies (*spawn policies*) spawn a new task when the program structure indicates that flow of control is likely to reach that task. For example, a loop branch will usually be taken and lead to the next loop iteration. Similarly, control flow is likely to reconverge at the static instruction following a procedure call.

Spawning tasks that begin at the *immediate postdominators* [7] of branches generalizes these heuristic techniques. The immediate postdominator of a branch is, roughly, the first instruction that is guaranteed to be fetched, no matter which way the branch resolves. Thus, a branch instruction's immediate postdominator is *control equivalent* to the branch itself. If the path leading up to the branch was correctly predicted then the immediate postdominator will also be fetched. Clearly, creating a new task at the immediate postdominator of a branch provides options for fetch that are no more speculative than the instructions fetched along the path that led to that branch.

The central insight underlying our spawn policy is that fetching the control flow graph of a program in program order is overly conservative. Unfolding the *control dependence graph* [7] of the program gives a less conservative fetch order [10]. When unfolding a control flow graph one fetches along the flow path defined by program order, which must be done speculatively since few branch directions are known at fetch time. When unfolding a control dependence graph one can fetch any of the blocks that are postdominators of recently fetched branches, providing control equivalent fetch opportunities. For this reason, we refer to task selection based on postdominance as *control equivalent spawning*.

The contributions of this paper include, first, a demonstration that *using all available immediate postdominance information to identify potential tasks is critical to achieving high performance in a speculative parallelization system*. Heuristics approximate only a subset of the postdominance information. Using compiler generated postdominator information provides an exhaustive set of potential tasks (*spawn points*), consisting of not only the tasks identified by heuristics, but also other tasks not easily obtained from heuristics. Selection of tasks based on immediate postdominator analysis directly translates into superior performance over policies that select tasks based on individual heuristics or combinations of heuristics.

Second, *we categorize the tasks identified by postdominator analysis and characterize the benefits of each task*

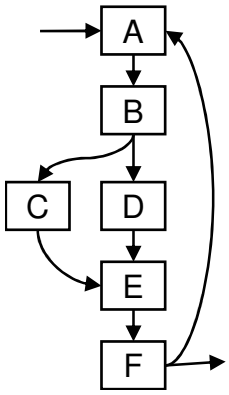

Figure 1. Control flow graph for a loop containing an if-then-else statement.

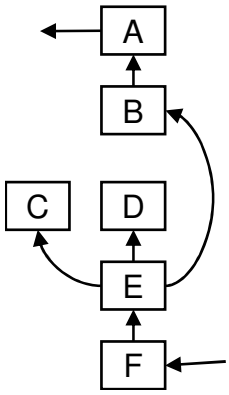

Figure 2. Postdominator tree for the flow graph from Figure 1. The parent of each node is its immediate postdominator.

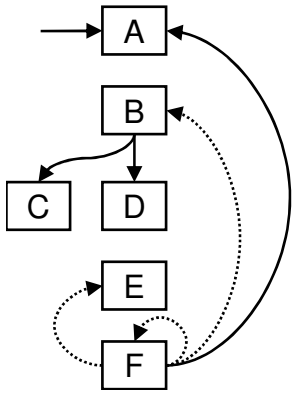

Figure 3. Control dependence, graph for the flow graph from Figure 1. Control dependence graph edges that are not also flow graph edges are shown with dotted lines.

type. The tasks identified by postdominator analysis can be broadly classified as simple hammocks, loop fall-throughs, procedure fall-throughs, or other control-flow. Each type of task extracts performance from different sources, such as reducing the penalty of branch mispredictions, initiating instruction cache misses, or exploiting outer-loop parallelism. Further, we find that it is critical to exploit all potential tasks identified as the postdominators of branches; excluding even one type of task impacts performance.

Third, for systems that don't have access to compiler generated information, *we illustrate that a dynamic mechanism can identify a set of potential tasks that closely approximates the set identified by immediate postdominator analysis*. For this purpose, we use a dynamic reconvergence predictor [5] that monitors the run-time retirement stream and is able to predict reconvergence information for branch instructions. A system using this scheme to support control-equivalent spawning is able to identify most control-equivalent tasks, and approaches the performance of a compiler-aided system.

The next section describes how to identify control-equivalent tasks using the immediate postdominators of branch instructions. This paper evaluates this spawn policy in the context of PolyFlow, a system designed to minimize the amount of speculation needed to support aggressive automatic parallelization. From a single sequential thread, PolyFlow spawns multiple, control-equivalent tasks that execute concurrently on a simultaneously multithreaded core. While PolyFlow's tasks are no more speculative than the fetch stream from which they were spawned, the tasks may be data-dependent on one another. In keeping with the notion of parallelization without unnecessary speculation, PolyFlow handles inter-task data dependences (through both memory and registers) in a conservative and complexity-effective fashion, without any value prediction or selective re-execution. Details of the PolyFlow architecture are described in Section 3.

Section 4 evaluates the advantages of control equivalent spawning. Related work in task selection policies for both compiler-based and dynamic speculative parallelization systems is described in Section 5, as well as previous work in exploiting control independence in superscalar processors. Finally, Section 6 draws insights and conclusions.

2 Control Equivalent Spawning

This paper argues that speculative parallelization systems should use the immediate postdominators of branching instructions as spawn opportunities. The next subsection defines postdominators and motivates their usefulness by explaining the sense in which the immediate postdominator of a branch is control equivalent to the path leading up to the branch. Section 2.2 gives more insight into the types of tasks identified by postdominator analysis, their relative merits, and the sources of performance they exploit. As an example, Section 2.3 provides details about the kinds of spawn opportunities available in an important loop from the SPEC2000 `twolf` benchmark. Finally, Section 2.4 describes a dynamic mechanism that closely approximates the set of tasks identified by compiler-based postdominator analysis.

2.1 Postdominance and Control Dependence

In a control flow graph (CFG), a node `i` *dominates* node `j` if every path from the entry node to `j` passes through `i` [7]. Postdominance information is obtained by finding dominators in the reversed CFG, with the entry and exit nodes interchanged along with the direction of all edges. That is, node `d` postdominates `i` if each path from `i` to the exit passes

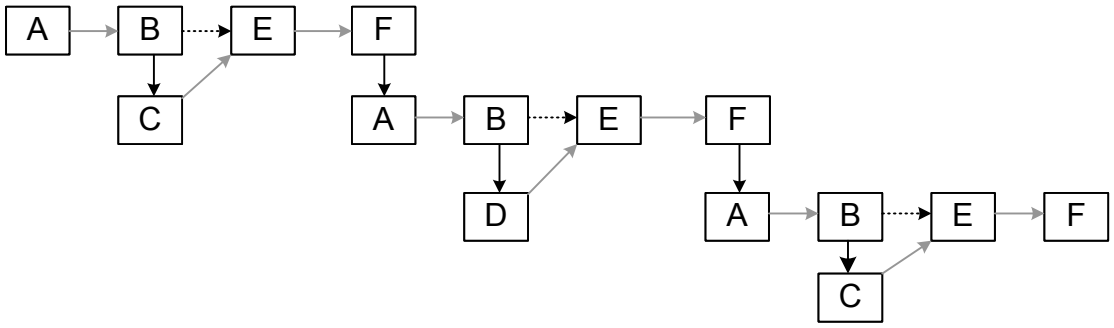

Figure 4. A possible dynamic fetch ordering for a run of the flow graph shown in Figure 1. Flow edges are shown solid, while spawns of immediate postdominators are shown dotted. Speculative fetches are shown in black. Non-speculative control flow graph edges are shown in gray. Program order runs left to right except that the flow moves downward every time the system speculates through a branch (the black edges flowing out of nodes B and F). Blocks A, B, E and F are control equivalent because every time the branch from F to A is correctly predicted all four blocks will execute.

through d. Stated another way, *whenever a CFG node executes, the flow of control is guaranteed to reach its postdominators*. Also, when d strictly postdominates i (*i.e.*, d postdominates i and $d \neq i$) and there is no node except d that strictly postdominates i on the paths from d to i, then d immediately postdominates i. Consider, for example, the flow graph shown in Figure 1. The flow graph represents six basic blocks organized as a loop containing an if-then-else statement. Figure 2 illustrates the postdominator tree for that flow graph. In this tree, for example, E postdominates B because control flow is guaranteed to reach E whenever it reaches B.

Control dependence is closely related to postdominance. Informally, a control flow graph block is control dependent on a branch if that branch is involved in determining whether or not the block executes. Thus, a block X is control dependent on a branch b if, of the two paths flowing from the branch, one includes X on all paths to the exit, and the other may flow to the graph exit without including X.

The control dependences for the flow graph in Figure 1 are shown in Figure 3. For example, blocks A, B, E and F are all control dependent on the loop branch in block F, while block E is not control dependent on either B, C or D, because all paths from B, C and D to the exit flow through E. Thus, the control dependence graph indicates instructions that are certain to execute when a particular branch decision is made. For example, every time the loop branch is taken from block F to block A, then blocks A, B, E and F (all from the next iteration) will execute. Note, further, that among the four blocks that are control dependent on the loop branch, A always flows directly to B and E always flows directly to F. While node B does not flow directly to E, E is the immediate postdominator of B.

Figure 4 shows one possible set of fetch choices that control equivalent spawning could select for the fetch unit. The "degree of speculation" runs from top to bottom in Figure 4. In this example the execution path through the flow graph consists of three iterations of the loop, following the blocks in the sequence A, B, C, E, F, A, B, D, E, F, A, B, C, E, F.

When block B is fetched, the spawn mechanism has the choice of spawning block E (the immediate postdominator of the branch in block B). In the example shown, the mechanism chooses to spawn E each time B is fetched, thereby allowing the machine to simultaneously fetch from any of the blocks that are control dependent on recently speculated branches. This example demonstrates that unfolding the control dependence graph provides a speculatively parallelizing architecture with many control-equivalent choices to augment the next block in the control flow.

One way to obtain the spawn edges described above is through a postdominator analysis of the program control flow graph. Whenever the control flow reaches a CFG basic block, it is guaranteed to reach the postdominators of that block, assuming that control is on the correct path. To exploit control-equivalent spawn opportunities, the system must be aware of each branch's immediate postdominator. Augmenting the program binary with compiler-generated postdominator information associated with each branch is one possibility. The PolyFlow speculative parallelization system dedicates a special cache for storing the addresses of the immediate postdominators of branches (much like a BTB stores branch targets), and has a separate section in the binary that is loaded into this cache on demand.

2.2 Understanding Postdominator Spawn Points

This section gives more insight into the types of tasks identified by postdominator analysis, their relative merits, and the sources of performance they exploit. Figure 5 classifies immediate post-dominators into four categories: loop fall-throughs, procedure fall-throughs, simple hammocks, and the rest, called "others" here.

Note that the classification is restricted to basic blocks that end in conditional branches. About one third of all basic blocks actually do not end in conditional branches. For

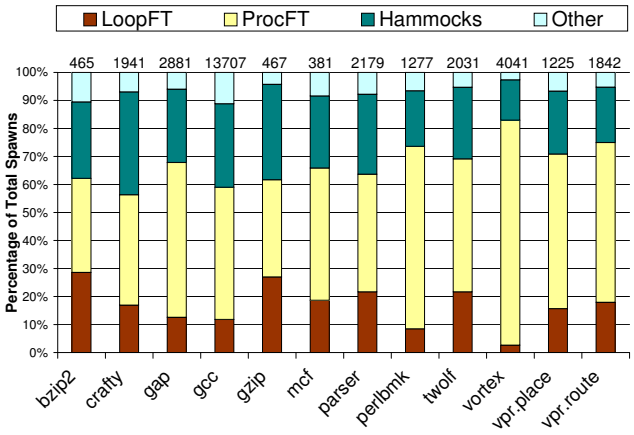

Figure 5. Static distribution of control-equivalent task types. Hammocks, loop fall-throughs and procedure fall-throughs are all important task types. The "other" category corresponds to unclassified branches including indirect jumps. The total number of static spawns is shown on the top of each bar.

example, in Figure 1, block A simply falls through to block B, rather than branching. The immediate postdominators of non-branching blocks are not good spawn points since the fetch unit will soon fetch those successor blocks along the conventional control-flow path.

The loop fall-through, procedure fall-through, and hammock categories correspond to the programming constructs of loops, procedure calls, and if-then (or if-then-else) statements, respectively. *Loop fall-through* spawns are the immediate postdominators of loop branches, including breaks and other exit conditions. *Procedure fall-throughs* are the immediate postdominators of call instructions.

Loop fall-throughs and procedure fall-throughs have been widely used as tasks in several speculative parallelization systems, identified either by a compiler [8, 11], or through dynamic heuristics [1]. Section 4 shows why loop and procedure fall-throughs are important types of tasks. Loop fall-throughs of inner loops expose parallelism in outer loops. Both kinds of spawns discover parallelism in relatively distant regions of the code and also initiate instruction cache misses.

The hammock class consists of tasks that begin at the postdominators of simple if-then and if-then-else statements. The postdominators are the joins of the two potential paths through the statement. Spawning this type of task, which most speculative parallelization systems have not directly exploited, often allows execution to jump over hard-to-predict branches, reducing the number of misspeculated instructions fetched.

Finally, there are a number of "other" tasks that do not easily map to any of the above programming constructs. Many of these tasks are derived from postdominators that incorporate complex control flow that is hard to identify through heuristics; some are also the immediate postdominators of indirect jumps. Section 4 shows that the system must exploit these tasks to achieve maximum performance gains.

2.3 Obtaining Loop Spawns from Postdominators

Another important class of tasks identified by most speculative parallelization systems are loop iterations. The insight behind loop spawns is that the control flow at the start of a loop iteration is likely to reach the next iteration (loop branches are usually taken), and so it might be beneficial to spawn the next iteration as a new task.

Control-equivalent spawning captures most important loop spawns through a combination of hammock and loop fall-through spawns. Hammock spawns allow tasks to jump over control flow embedded within a loop iteration, ultimately reaching the basic block that contains the loop branch. Loop fall-through spawns allow the system to fetch beyond the inner loop branch, exposing parallelism in outer loops.

`Twolf` is a benchmark in which loop spawns yield significant speedups, and thus provides more insight into how control-equivalent spawning can capture the benefits of loop spawns. The function `new_dbox_a`, consisting of a nested for-loop, is shown in Figure 6. The inner loop consists of a linked list traversal. It contains an if-then-else statement, and two if-then statements (corresponding to the ABS macro). These blocks have hard-to-predict branches, in particular the if-then-else branch is taken about 30% of the time while the two if-else branches are taken about 50% of the time. The inner loop executes three iterations on average.

For the purposes of spawning a loop iteration it is better to spawn the last basic block of the loop (which ends in the loop branch) from the loop entry, as opposed to spawning the start of next loop iteration from the start of current loop iteration. Since the loop index variable increment tends to happen just before the loop branch, spawning the last basic block eliminates the data dependence on the index variable between different tasks. For the above example, instead of PC `9d60` spawning itself as the next loop iteration, the system spawns PC `9dec` whenever it reaches PC `9d60`. Note here that the index variable `termptr` is loaded from memory just before the loop branch (`9f2c`). Spawning PC `9d60` makes this important instruction local to a task, thus making the task independent from a dataflow perspective.

When spawning only loop iterations the most frequently performed spawn in `twolf` is the inner loop spawn (`9da0` $\rightarrow$ `9dd8`). The outer loop spawn (`9d60` $\rightarrow$ `9f28`) also happens quite frequently. Control-equivalent spawning can find the same opportunities through a combination of hammock spawns and loop fall-through spawns. In order to get inner loop iteration spawns, control-equivalent spawning needs to do three hammock spawns ((`9da0` $\rightarrow$ `9dbc`,

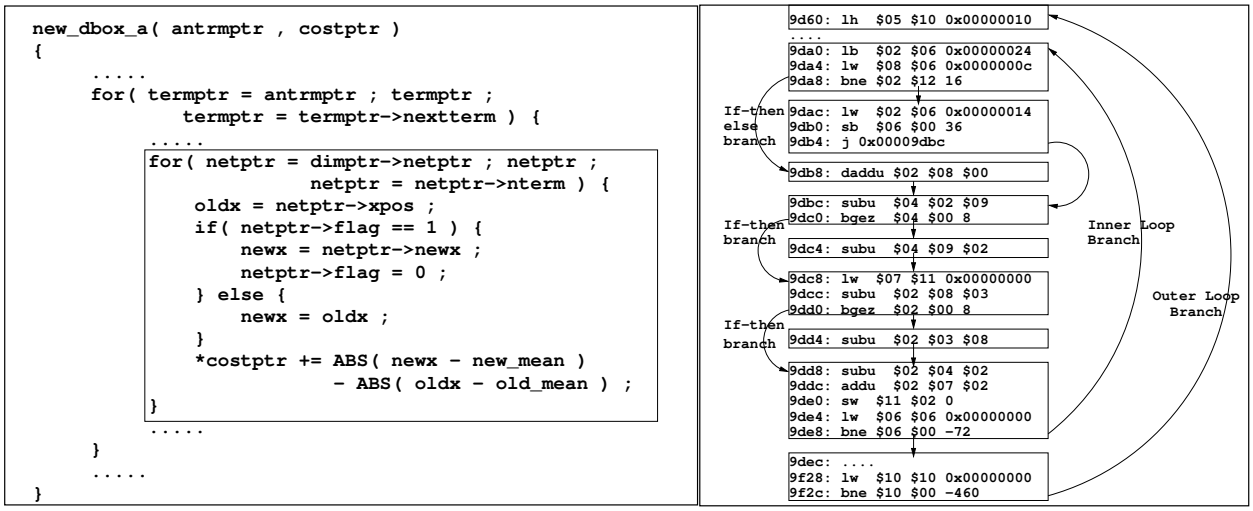

Figure 6. An important region in `twolf`. The high-level code is shown on the left. The assembly code, with the CFG overlaid, is shown on the right. The important loop spawns are `9da0` → `9dd8` (inner loop spawn) and `9d60` → `9f28` (outer loop spawn). The loop fall-through spawn at `9dd8` → `9dec` effectively spawns the next outer loop iteration. The hammock spawns at `9da0` → `9dbc`, `9dbc` → `9dc8`, and `9dc8` → `9dd8` can combine to spawn the next loop iteration.

`9dbc` → `9dc8`, and `9dc8` → `9dd8`). Experimentally, these spawns are actually the three most frequent spawns performed by a hammock-only spawn policy. The outer loop iteration spawn is obtained from the fall-through of the inner loop (`9dd8` → `9dec`), which is the most common spawn performed in `twolf` by a loop fall-through only spawn policy. The performance results shown in Section 4.1 indicate that loop fall-through spawns and hammock spawns perform similarly, or better than, loop spawns on `twolf`.

This example demonstrates that loop fall-throughs spawn outer loop iterations, thus exploiting outer loop parallelism, while hammock spawns jump across the hard branches inside the inner loop, leading to spawns of inner loop iterations.

2.4 Dynamic Mechanisms to Learn Spawn Opportunities

One of the drawbacks of the compiler-assisted techniques for task selection is that they require a change to the ISA or binary formats. Therefore, a dynamic mechanism to reconstruct postdominator information at run-time seems an attractive alternative. On the other hand, postdominance is computed through a backwards analysis in the compiler, and approximating it dynamically through a forward analysis might miss out on valuable spawn points. This section explores the trade-offs involved in identifying postdominators dynamically.

Dynamic reconvergence prediction [5] is a previously proposed dynamic mechanism for predicting control flow reconvergence information for each branch. A run-time predictor learns the reconvergence behavior for each branch, and classifies it into one of four categories. This predictor can then be used to obtain the reconvergence point for a branch, which is the point where control flow is expected to reconverge for the branch. This reconvergence point approximates the immediate postdominator of the basic block containing the branch instruction.

A reconvergence predictor profiles the instructions committed by a program to identify and update a candidate reconvergence point for each of the possible four categories for a targeted branch. Additional information is collected which allows accurate prediction of the reconvergence point by selecting among these four possibilities. The most important of these four categories consists of branches whose reconvergence PC is below the branch PC itself in the program layout. This category captures, among others, forward branches corresponding to if and if-else statements, as well as backward loop branches.

Section 4 shows that using the reconvergences learned by the reconvergence predictor as spawn points is beneficial. At each branch instruction, the system can spawn the reconvergence point of that branch as a new task. Section 4.4 shows that the dynamic reconvergence predictor is able to approximate the immediate postdominators of branch instructions reasonably well. Warm-up effects and hard-to-identify reconvergences are the main difference between the two.

3 Implementation and Methodology

The PolyFlow architecture uses control-equivalent spawning to speculatively parallelize sequential programs into parallel tasks that execute concurrently on a simulta-

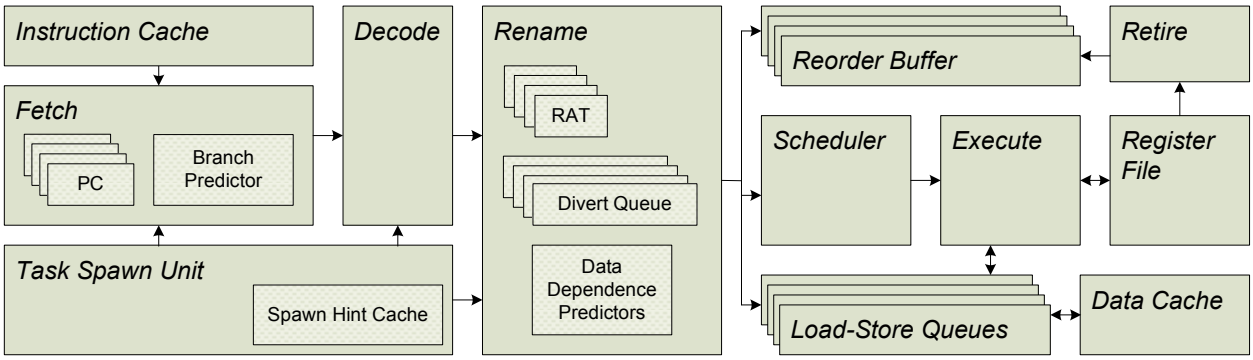

Figure 7. PolyFlow Machine Microarchitecture. PolyFlow is a speculative parallelization system built on a simultaneously multithreaded core. Immediate postdominator information of branches is stored in the spawn hint cache. The data dependence predictors in the rename stage are used to identify spawned consumer instructions that are dependent on producer instructions from the spawning task. The divert queue is used to delay renaming these consumer instructions until the producer instruction is dispatched to the scheduler [12, 20].

neously multithreaded core. PolyFlow uses novel mechanisms to handle inter-task data dependences (through both memory and registers) in a conservative and complexity-effective fashion, without any value prediction or selective re-execution. The next section provides a brief description of the PolyFlow architecture, while Section 3.2 discusses the execution-driven simulator infrastructure and evaluation methodology.

3.1 The PolyFlow Architecture

PolyFlow's machine architecture, shown in Figure 7, is similar to a simultaneously multithreaded (SMT) processor. PolyFlow incorporates a hardware structure called the Task Spawn Unit that is responsible for spawning new tasks. The Task Spawn Unit contains a "hint cache" that associates control-equivalent spawn points with branch PCs. When a task fetches a PC that has a spawn point associated with it, the Spawn Unit may decide to spawn the new task, depending on dynamic feedback about which tasks are profitable. The hint cache also contains an eight byte entry per spawn point, which is used to store register and memory dependence information for the task.

Like a simultaneously multithreaded processor, PolyFlow's backend has a unified scheduler. Similar to the POWER5 [17], entries in the reorder buffer are dynamically shared among active tasks. PolyFlow *synchronizes* data dependences between tasks. In particular, instructions that are predicted to depend on register or memory values that will be (but have not yet been) produced by an earlier task are not allowed to execute. Instead, these instructions (called *diverted* instructions) are stored in a FIFO structure called the *divert queue*. A diverted instruction is removed from the divert queue and dispatched into the scheduler some time after its corresponding producer instruction has been dispatched. Data-dependence violations lead to squashes of the violating task, as well as all tasks beyond it. More details about the hardware mechanisms used to identify such instructions and ensure correctness can be found in the work by Stone et al. [20] and Malik [12].

3.2 Simulation Infrastructure and Methodology

All experimental evaluation was performed on a fully execution-driven simulator running a variant of the 64-bit MIPS instruction set ISA. The ISA does *not* have any special instructions to support multithreading. It not only simulates timing, but also *executes* instructions out-of-order in the backend, writing results to the register file out of program order. When an instruction is retired, its results are compared against an architectural simulator, and an error is signaled if the results do not match. When a branch mispredict is discovered, the simulator immediately reclaims backend resources (ROB and scheduler entries, etc.) and recovers using a rename checkpoint.

The simulator obtains its spawn points from a profile-driven immediate postdominator analysis, and the Task Spawn Unit uses a trace to ensure that tasks are not spawned too far into the future. Conflict- and capacity- miss effects are not modeled in the hint cache. The Task Spawn Unit allows spawning only from the tail task, which is the youngest (most recently spawned) task in the machine.

The processor simulated is an aggressive, 8-wide machine, running up to 8 tasks, with the configuration given in Figure 8. Both PolyFlow's underlying SMT and the baseline superscalar use the same hardware resources. The superscalar is capable of fetching a maximum of one taken branch per cycle, while PolyFlow can fetch from two tasks in a cycle, with a maximum of one taken branch per cycle per task. The instruction fetch unit uses biased ICount to prioritize among different tasks [24]. The physical register file is large enough so as to never cause a stall.

The results presented are from running the SPEC2000 integer benchmarks, each with the lgred or mdred Min-

Parameter	Value
Pipeline Width	8 instrs/cycle
Branch Predictor	16Kbit gshare, 8 bits of global history
Misprediction Penalty	At least 8 cycles
Reorder Buffer	512 entries, dynamically shared
Scheduler	64 entries, dynamically shared
Functional Units	8 identical general purpose units
L1 I-Cache	8Kbytes, 2-way set assoc., 128 byte lines, 10 cycle miss
L1 D-Cache	16Kbytes, 4-way set assoc., 64 byte lines, 10 cycle miss
L2 Cache	512Kbytes, 8-way set assoc., 128 byte lines, 100 cycle miss
Divert Queue	128 entries, dynamically shared
Tasks	8

Figure 8. Pipeline parameters.

nesota Reduced inputs. Results for `eon` are not shown since our tool chain cannot compile c++ benchmarks. The simulator fast-forwards through the initialization phase of all benchmarks, and executes 100 million instructions after that. With the exception of Figure 11, all performance graphs show the speedup of different speculative parallelization configurations over a superscalar with equivalent machine resources. The IPCs attained by the superscalar are shown below the *x*-axis in Figure 9.

4 Evaluation

This section evaluates the performance of control-equivalent spawning and a variety of heuristic spawn policies. The results demonstrate that applications vary widely in their response to individual heuristics, and that control equivalent spawning subsumes all important heuristics. Most speculative parallelization systems use some combination of heuristics to identify spawn points, and Section 4.2 shows that using postdominators as spawn points performs better than existing combinations of heuristics. Section 4.3 investigates the relative importance of different types of tasks identified by immediate postdominator analysis, and shows that excluding any class of tasks can hurt performance. Finally, Section 4.4 presents the performance of a dynamic mechanism that approximates postdominator analysis.

4.1 Individual Heuristic Spawn Policies

This section evaluates the relative importance of the following types of spawns: loops, loop fall-throughs, procedure fall-throughs, hammocks, and "other." Figure 9 evaluates each of these spawn types in terms of PolyFlow's speedup over a superscalar processor.

Loops, loop fall-throughs, and procedure fall-through heuristics are widely exploited in speculative parallelization, and Figure 9 confirms that they are indeed important sources of spawn points. For example, `vortex` and `gap` perform well with procedure fall-through spawns, which reduce stalls from instruction cache misses. As described in Section 2.3, `twolf` contains inner- and outer-loop parallelism; thus it performs well with both loop and loop fall-through spawns.

Note that loop spawns do not perform as well as other heuristics in PolyFlow, largely because PolyFlow's SMT backend is already quite effective at exploiting inner loop parallelism within a single thread. Hence, unless there is hard-to-predict control flow inside the loop, there are limited gains from spawning loop iterations as separate tasks. Rather, loop spawns consume task contexts and tend to create data dependences between tasks.

Hammock spawns speed up the execution of most applications. As explained in Section 2.2, hammocks can effectively capture several important loop spawns. In addition, they can capture spawn opportunities corresponding to any construct (such as procedure calls or loops) that is embedded inside if-then or if-then-else blocks. By jumping over hard to predict branches, hammock spawns speed up `mcf` and `crafty`, where other heuristics had little impact. Finally, the "other" category is also valuable. In particular, in `perlbmk`, "other" spawns are better than the remaining heuristics. Given that "other" spawns account for a small number of static spawn points and a small number of dynamic spawns, this result implies that "other" spawns, when they exist, tend to create parallelism efficiently.

While individual heuristics do well in specific applications, there is no single heuristic that captures all or most of the important spawn points. On the other hand, spawning from immediate postdominators of branches captures all of these important sources. Control-equivalent spawning either outperforms or comes close to the best individual heuristic for each individual benchmark.

4.2 Heuristic Combination Policies

As described in Section 5, several speculative parallelization systems obtain their spawn points from a combination of loops, loop fall-throughs, and procedure fall-throughs. Figure 10 compares proposed combinations of heuristic policies. These heuristic combinations are compared to control equivalent spawning. The performance edge of the latter continues to hold over these combination heuristics.

On PolyFlow, using control equivalent spawning performs at least as well as the best heuristic combination policy. In several cases, such as `crafty` and `mcf`, it significantly outperforms the best considered heuristic policy.

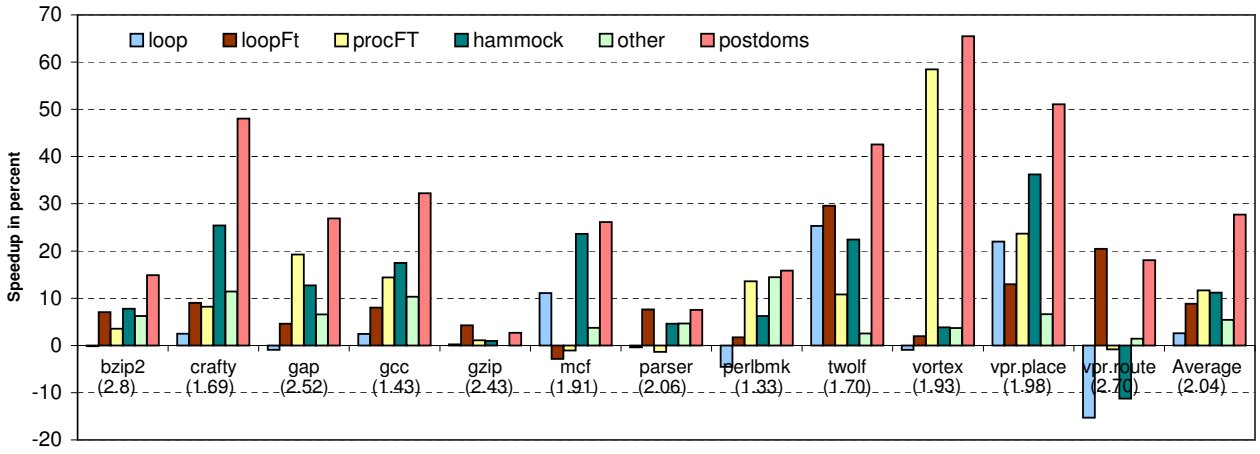

Figure 9. Individual Heuristic Policies for Spawn Points. Speedups shown are over an 8-wide superscalar, with superscalar IPCs reported below each benchmark. Control-equivalent spawning covers all considered heuristic policies, and on an average, more than doubles the speedup of the best heuristic.

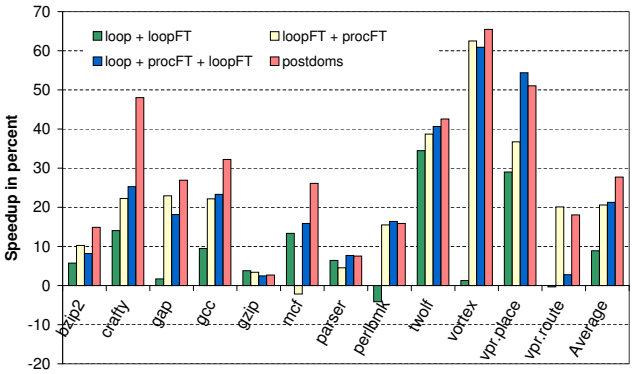

Figure 10. Combination of Heuristics for Spawn Points. Spawning immediate postdominators is superior to other widely used combinations of loop, loop fall-through, and procedure fall-through spawn points.

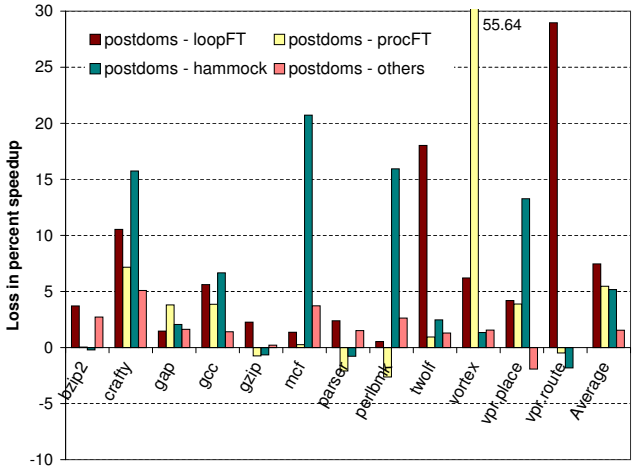

Figure 11. Loss in speedup compared to spawning from the full postdominator set (normalized to Superscalar IPC), for policies that exclude spawns from a single category. Using full immediate postdominator information for spawning is critical to performance.

4.3 Importance of Spawning All Postdominators

This section demonstrates that each of the four major categories of immediate postdominators (loop fall-throughs, procedure fall-throughs, hammocks, and "other") are valuable and necessary to exploit the full potential for parallelization. Figure 11 shows the loss in speedups for policies which lack just one type of spawn, compared to using all types of immediate postdominators for spawning. These losses are normalized to the observed Superscalar IPCs.

The figure indicates that missing out on even one of these categories can hurt performance in specific benchmarks where they uncover important opportunities. For instance, `vpr.route` suffers a 29% loss when loop fall-through spawns are removed. `Vortex` takes a 56% hit when procedure fall-throughs are removed from the spawn point set. `Perlbmk` and `mcf` lose 21% and 16% respectively when hammocks are removed. Finally, the statically small set of "other" postdominators is also valuable, as evidenced by the performance drop in `crafty`, `mcf` and `perlbmk` when these spawn points are removed.

Occasionally a spawn policy that restricts the set of spawns will achieve a small (less than 2%) improvement in performance over the full immediate postdominator set. These improvements happen when a benchmark is particularly receptive to a certain type of spawns (*e.g.*, `vpr.route` and loop fall-throughs, as seen in Figure 9). Since the number of tasks that the system can run simultaneously is limited, adding other types of spawn points results in fewer spawns of the receptive kind, thereby causing a small reduction in performance.

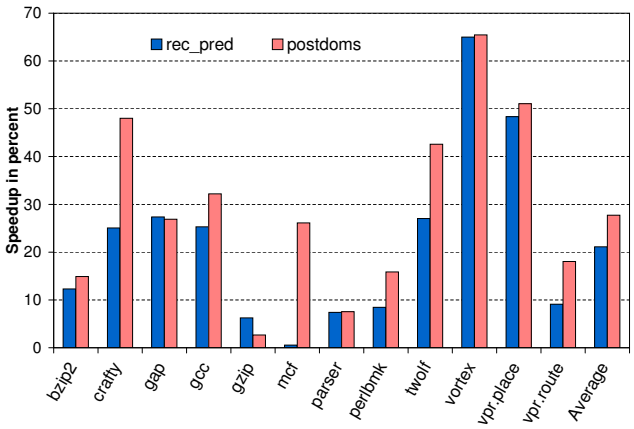

Figure 12. Spawning using Reconvergence Prediction. Spawning the reconvergence point of branches, which approximates immediate postdominator information, gets close to spawning from compiler-generated immediate postdominator information, and can be an effective dynamic task selection technique.

4.4 Using Reconvergence Predictor to Reconstruct Postdominator Information

This section investigates a dynamic mechanisms that approximates compiler-based postdominator analysis. The dynamic reconvergence predictor [5] predicts, for each branch, a reconvergence point where control flow is likely to reconverge. This reconvergence point approximates the immediate postdominator of the basic block containing the branch instruction. To achieve the best approximation possible, we implement the most aggressive reconvergence predictor described in [5], and train it on the retirement stream obtained from the reorder buffer. The trained predictor is then used to obtain spawn points. Thus, the simulator models warm-up effects, which incorporate the time taken to learn reconvergences for branches. The simulator does not, however, model capacity and conflict effects in the cache structure used to store the reconvergence information.

The reconvergence predictor is used to obtain spawn points as follows. Upon reaching any branch, the system identifies the reconvergence point of that branch as a possible spawn point. This reconvergence point approximates the immediate postdominator of the basic block containing the branch. In addition, the system also spawns procedure fall-throughs at call instructions. Figure 12 compares the performance of a spawn policy using the reconvergence predictor to one that utilizes compiler generated immediate postdominators (in terms of speedups over the superscalar processor).

While this dynamic scheme performs quite well and gets close to the compiler-aided system, it lags behind appreciably in several cases, notably `crafty`, `mcf` and `twolf`. A significant portion of this loss is due to warm-up effects, where the inability to spawn a task the first few times misses out on valuable opportunities. Also, since the reconvergence predictor is a dynamic mechanism, it is hard for it to capture all the immediate postdominators, and the rest of the performance difference lies there. On the whole, however, we find that the reconvergence predictor approximates the immediate postdominator information with reasonable accuracy.

5 Related Work

Task identification for speculative parallelization has been approached in multiple ways, varying from compiler identification of tasks, to dynamic heuristic-based task creation. This section summarizes some of the important related work in this area.

Many compilers for thread-level speculation (TLS) on shared-memory multiprocessors [3, 6, 21, 25] rely on loops as candidates for parallel execution, and loop iterations are the only possible tasks. Loop unrolling, and loop interchange are applied in conjunction with task selection to create tasks of suitable sizes. Some of these compilers support task selection policies that include procedure fall-throughs in addition to loops [8, 11]. The POSH compiler [11] tries to hoist these loop, loop fall-through, and procedure fall-through spawn points as high as possible, using control equivalence along with other constraints on data dependence and task spawn ordering. Section 4 showed that spawning from postdominators captures all of these individual heuristic spawns.

Several systems try to use profiled path information to create speculative tasks along the frequently executed paths. Control Quasi-Independence (CQIP) uses profile information to reconstruct the dynamic program control flow graph with edges weighted by execution frequency [14]. Basic block pairs that are likely to lie on the same path are identified as possible spawning point and control quasi-independent points. Bhowmik et al. [2] also describe a compiler system that uses path profiles to identify tasks. It starts out by trying to create tasks out of loop iterations. Next, it tries to create tasks along common paths, as well as infrequent paths, for each immediate postdominator pair. Control equivalent spawning, in contrast, creates new threads that are no more speculative than their spawn point, by exploiting the control equivalence property of immediate postdominators.

The Multiscalar compiler [23] performs task selection by walking the static program CFG, and partitioning it using a variety of heuristics, which incorporate task size, inter-task control flow, and inter-task data dependence. It uses loop unrolling and function inlining to make tasks of appropriate size. Tasks are not allowed to cross loop or function entries or exits. The CFG edges can also be annotated with edge weights and partitioned into tasks using a min-cut algorithm [9].

The Dynamic Multi-Threading (DMT) processor [1]

uses dynamic heuristics to spawn at procedure and loop fall-throughs. It approximates loop fall-throughs by spawning the static address directly following each backward branch. A history buffer is used to predict after-loop thread addresses that differ from this default value. A subsequent work [19] implements a run-ahead policy that also spawns the instruction following an L3 cache miss.

The clustered speculative multithreaded processor uses a dynamic loop detector to identify and spawn loop iterations as tasks [22]. Such loop-iteration based spawns have been found preferable to loop fall through or procedure fall through based spawns in the context of the clustered speculative multithreaded processors [13]. This paper's static and dynamic techniques capture more spawn opportunities than both of the above systems.

Dynamic reconvergence prediction [5] is a dynamic mechanism for predicting control flow reconvergence information for each branch. This predictor has been used in DMT to spawn threads at the reconvergence points of indirect jumps, which leads to speedups of up to 18% and average speedups of 4% over the baseline, beyond the speedups achieved by DMT. This paper builds on this work to identify more types spawn opportunities, and in the process builds an effective dynamic spawn identification mechanism.

Research has also been done in exploiting control independence in the context of superscalar processors. The Skipper [4] processor exploits control independence to skip instructions control dependent on hard to predict branches. Rather it fetches and executes instruction control independent of those branches. Skipper uses heuristics to predict the reconvergence points of branches corresponding to if-then and if-then-else constructs.

Lam and Wilson's limit study [10] showed that exploiting control independence to fetch and execute along multiple flows of control can expose large amounts of instruction level parallelism, which is not possible for a superscalar processor limited by branch prediction accuracy.

Rotenberg et al. explored exploiting control independence to increase performance of superscalar processors [15] based on the insight that instructions control independent of a branch are executed irrespective of the direction of the branch. They demonstrate potential speedups of up to 30% on a wide superscalar processor. Squash reuse [16, 18] is another approach that exploits control independence to salvage work in control independent regions upon a branch mispredict.

6 Conclusions

This paper demonstrates that identifying spawn points using the immediate postdominators of branch instructions is critical to achieving high performance in a speculative parallelization system. Heuristics based on programming language constructs such as loops and procedure calls fail to capture valuable spawn points. This paper shows that control-equivalent spawning achieves, on average, more than twice the speedup of the best individual heuristic and 33% more speedup than the heuristic policy that uses loops, loop and procedure fall-throughs. Further, it demonstrates that dynamic reconvergence prediction [5] provides a close approximation to compiler-generated immediate postdominance information, and can be used as an effective dynamic mechanism for identifying spawn points.

Task selection based on immediate postdominator analysis provides a variety of benefits; most notably, this technique extracts control-equivalent tasks and exposes outer loop parallelism. The current version of the PolyFlow architecture has relatively limited resources available to fully exploit these possibilities. For example, the current system allows each thread to spawn only a single successor, so PolyFlow can spawn only the outer-most branch of a nested if-then-else construct and is unable to spawn past the branch in the inner hammock. In addition, because the reorder buffer has only 512 entries, and is unable to reclaim resources from younger threads, PolyFlow can exploit only a limited amount of outer loop parallelism. We hope to address both of these limitations in future work so that the PolyFlow system can deliver even better parallelism.

Acknowledgments

We are grateful to several people for helping to make this paper possible. Sanjay Patel was an enthusiastic contributor to the early formulation of this work. Todd Rafacz introduced us to dynamic reconvergence prediction and contributed an initial implementation. Amir Roth and Steve Lumetta expressed interest in our discussions about control-equivalent spawning and encouraged us to write up the work. Vikram Dhar provided indispensable help with figures and graphs. Craig Zilles provided valuable feedback on an early draft of the paper.

The work reported in this paper was supported in part by the National Science Foundation under grant CCR-0429711 and by the MARCO Gigascale Systems Research Center. Sam Stone is supported under a National Science Foundation Graduate Research Fellowship. Computational resources were supported by an equipment donation from AMD Corp. and the National Science Foundation under grant EIA-0224453, and through the contribution of the use of a computing cluster by the Trusted ILLIAC Center at the Information Trust Institute and Coordinated Science Laboratory at the University of Illinois. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. *Int'l Symp. Microarchitecture*, (MICRO-31):226–236, 1998.

[2] A. Bhowmik and M. Franklin. A general compiler framework for speculative multithreaded processors. *IEEE Trans. Parallel Distrib. Syst.*, 15(8):713–724, 2004.

[3] P.-S. Chen, M.-Y. Hung, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. *Principles and Practice of Parallel Programming*, (PPoPP-9):25–36, 2003.

[4] C.-Y. Cher and T. N. Vijaykumar. Skipper: A microarchitecture for exploiting control-flow independence. *Int'l. Symp. Microarchitecture*, (MICRO-34):4–15, 2001.

[5] J. D. Collins, D. M. Tullsen, and H. Wang. Control flow optimization via dynamic reconvergence prediction. *Int'l. Symp. Microarchitecture*, (MICRO 37):129–140, 2004.

[6] Z.-H. Du, C.-C. Lim, X.-F. Li, C. Yang, Q. Zhao, and T.-F. Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. *Prog. Lang. Design and Implementation*, (PLDI):71–81, 2004.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. Syst.*, 9(3):319–349, July 1987.

[8] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. *Arch. Support Prog. Lang. Operating Sys.*, (ASPLOS-VIII):58–69, 1998.

[9] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. *Prog. Lang. Design and Implementation*, (PLDI):59–70, 2004.

[10] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. *Int'l. Symp. Comp. Arch.*, (ISCA-19):46–57, 1992.

[11] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: A TLS compiler that exploits program structure. *Principles and Practice of Parallel Programming*, (PPoPP-11):158–167, 2006.

[12] K. Malik. Confidence based out-of-order register renaming for dynamically multithreaded processors. Master's thesis, University of Illinois Department of Electrical and Computer Engineering, Dec. 2006.

[13] P. Marcuello and A. González. A Quantitative Assessment of Thread-level Speculation Techniques. *Int'l. Parallel and Distributed Proc. Symp.*, (IPDPS-14):595–604, 2000.

[14] P. Marcuello and A. González. Thread-spawning schemes for speculative multithreading. *High Perf. Comp. Arch.*, (HPCA-8):55–64, 2002.

[15] E. Rotenberg, Q. Jacobson, and J. Smith. A study of control independence in superscalar processors. *High Perf. Comp. Arch.*, (HPCA-5):115–124, 1999.

[16] A. Roth and G. S. Sohi. Register integration: a simple and efficient implementation of squash reuse. *Int'l. Symp. Microarchitecture*, (MICRO-33):223–234, 2000.

[17] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM J. Rsch. Dev.*, 49(4/5), 2005.

[18] A. Sodani and G. S. Sohi. Dynamic instruction reuse. *Int'l. Symp. Comp. Arch.*, (ISCA-24):194–205, 1997.

[19] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. *Arch. Support Prog. Lang. Operating Sys.*, (ASPLOS-XI):107–119, 2004.

[20] S. S. Stone, K. M. Woley, K. Malik, M. Agarwal, V. Dhar, and M. I. Frank. Synchronizing store sets (SSS): Balancing the benefits and risks of inter-thread load speculation. Technical Report CRHC-06-14, University of Illinois, Center for Reliable and High-Performance Computing, Dec. 2006.

[21] J.-Y. Tsai, Z. Jiang, and P.-C. Yew. Compiler techniques for the superthreaded architectures. *Int. J. Parallel Program.*, 27(1):1–19, 1999.

[22] J. Tubella and A. González. Control speculation in multithreaded processors through dynamic loop detection. *High Perf. Comp. Arch.*, (HPCA-4):14–23, 1998.

[23] T. N. Vijaykumar and G. S. Sohi. Task selection for a multiscalar processor. *Int'l Symp. Microarchitecture*, (MICRO-31):81–92, 1998.

[24] S. Wallace, B. Calder, and D. M. Tullsen. Threaded multiple path execution. *Int'l. Symp. Comp. Arch.*, (ISCA-25):238–249, 1998.

[25] A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry. Compiler optimization of scalar value communication between speculative threads. *Arch. Support Prog. Lang. Operating Sys.*, (ASPLOS-X):171–183, 2002.