Branch-mispredict Level Parallelism (BLP) for Control Independence

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

Abstract

A microprocessor's performance is fundamentally limited by the rate at which it can resolve branch mispredictions. Control independence *(CI) architectures look for useful control and data independent instructions to fetch and execute in the shadow of a branch misprediction. This paper demonstrates that CI architectures can be guided to exploit substantial* branch-mispredict level parallelism *(BLP) in existing control intensive applications. A program has branch-mispredict level parallelism when its dynamic execution trace contains hard-to-predict branches that are both control and data independent, and thus could, potentially, be resolved in parallel.*

Although applications have a high degree of inherent BLP, we find that the amount of BLP exploited by naive CI architectures tends to be quite small. We show that spawn selection and data dependence handling policies in a CI architecture should make choices that explicitly aim to maximize branch-mispredict level parallelism. We demonstrate that with BLP-focussed policies, CI architectures can expose high amounts of branch-mispredict level parallelism and achieve 50% to 90% improvements in IPC on several of the SPEC 2000 Integer benchmarks.

1 Introduction

Mispredicted branches, and the backwards slices of those branches, comprise a large fraction of the critical paths through control intensive programs [9, 4, 31, 15]. The mean time between branch mispredictions is a first order determinant of superscalar performance [15] because the branch misprediction feedback mechanism serializes branch misprediction resolution [4, 31]. All instructions that were fetched between the time a mispredicted branch is fetched and the time it is resolved must be flushed, even if those instructions performed, or will perform, useful work.

The impact of branch mispredictions can be mitigated by better branch prediction (reducing the number of branch mispredictions) or through multi-path execution [34] and predication [16] techniques that fetch useful work during the period while hard-to-predict branches are resolving. Control independence processors [28, 5, 13, 3] fetch only from a single path but find work along that path that is independent of a branch mispredict.

This paper explores overlapping the penalty of multiple mispredicts on control independence processors to alleviate the branch mispredict bottleneck. We demonstrate that the *performance of a control independence processor is strongly correlated to the average number of branch mispredictions that it resolves in parallel*. In analogy to "Memory-Level Parallelism" (MLP), where one searches for another cache miss to do in the shadow of a cache miss [24, 32] we call this overlapping of branch mispredict penalties "Branch-mispredict Level Parallelism" (BLP).

This paper introduces and formalizes the concept of Branch-mispredict Level Parallelism (BLP). We show that significant amounts of BLP exist in control-intensive applications, and that the BLP a control independence processor extracts from an application strongly correlates to the performance the processor achieves. While BLP can be exploited by a control independence (CI) architecture, we find find that BLP-centric spawn selection and data dependence handling policies can dramatically affect the amount of BLP exploited. On a 4-core setup, a BLP-centric system achieves 50% to 90% improvements in IPC on several of the SPEC 2000 Integer benchmarks, while a naive CI system achieves a maximum of 15% improvement.

The rest of the paper is structured as follows. The next section motivates branch-mispredict level parallelism. Section 3 describes techniques we use to find BLP. Section 4 explains our experimental methodology and presents experimental results. Finally, we conclude in Section 5.

2 Motivation

Serialized branch mispredictions bottleneck superscalar performance. In this section, we explore the potential of improving performance by resolving multiple independent mispredicted branches in parallel.

Section 2.1 explains how serialized branch mispredictions limit performance in conventional superscalar processors. Section 2.2 introduces the concept of BLP, and describes the conditions under which branch mispredicts can

978-1-4244-2070-4/08/$25.00 ©2008 IEEE

be parallelized. Section 2.3 describes related work in Control Independence (CI) architectures, which, unlike superscalars, can overlap the misprediction penalties of multiple branches.

2.1 The Branch Misprediction Bottleneck

The top half of Figure 1 depicts the resolution of consecutive mispredicted branches in a superscalar. Although the superscalar can execute several branches simultaneously and out-of-order, it can only resolve one misprediction at a time. Even if the processor executes two mispredicted branches (say A and E) in parallel, detection of the earlier misprediction (A) causes all subsequent instructions, including the mispredicted branch E, to be squashed, fetched and executed again.

This is shown in more detail in Figure 1. The timeline here refers to the flow graph in Figure 2(a). Basic block A ends in a branch. Shown at the top of Figure 1 is the stream of instructions fetched over time. The processor mispredicts the branch at the end of block A, and thus fetches block B, when it should have fetched block C. It then incorrectly fetches blocks E, F and H. Finally, the branch in block A executes, and the misprediction is detected. All instructions from blocks B, E, F and H are flushed and fetch is restarted at block C. Note, in particular, that an instance (labeled E') of block E has been fetched and possibly executed, but because this instance was squashed, block E needs to be fetched again, and the resolution of the misprediction in block E does not overlap the resolution of the misprediction in block A.

Given the impact of branch mispredictions on performance, it is productive to find useful work to do in the shadow of a branch misprediction [10, 30, 28, 5, 13, 3]. We demonstrate in this paper that the number of *mispredicted branches* resolved in the shadow of a mispredict strongly correlates to performance improvement. This approach is demonstrated in the bottom half of Figure 1. If the work being done to resolve the branch at block E is independent of the work being done in blocks A or C, then starting the resolution of the misprediction in block E early might increase performance. In Section 2.3 we describe how to use a speculative multi-threading system to start the resolution of block E early.

2.2 Branch-mispredict Level Parallelism

Branch-mispredict Level Parallelism (BLP) indicates the extent to which the misprediction penalties of different branches are overlapped. We say that a branch is *unresolved* from the time it is fetched to the time it gets executed. Analogous to memory-level parallelism (MLP), BLP is defined as the average number of independent (both from a control and data standpoint) mispredicted branches that are *unresolved*, when at least one mispredicted branch is unresolved. BLP takes into account only branches that successfully retire. For a superscalar, BLP is exactly 1. Just like MLP, different programs have different levels of BLP.

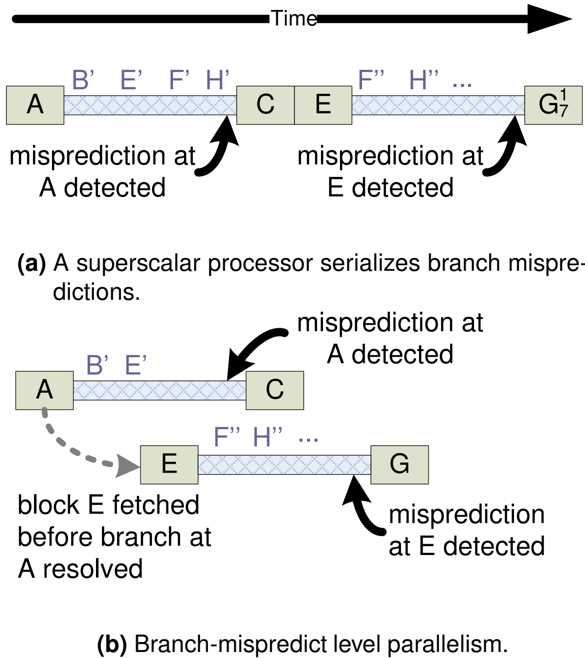
Figure 2 gives examples of the two basic conditions that must be true of mispredicted branches for them to be resolved in parallel: the branches must be control independent and data independent of each other. Consider the branch at block A in Figure 2(a). The branch at block E can be resolved in parallel with the branch at block A, while the branch at block B can *not* be resolved in parallel with the branch at block A. We say that block B is *control dependent* on the branch at block A because, informally, the branch at block A "decides" whether or not block B should execute. The branch in block E, on the other hand, is *control independent* of the branch in block A, because no matter which direction the branch at A goes, block E still needs to be fetched and executed. From a microarchitectural standpoint, the useful property is that if a block E *postdominates* block A, then block E is control independent of block A. We say that block E postdominates block A iff all paths from block A to the program exit pass through block E [8]. In Figure 2(a) block E postdominates block A, while B does not postdominate block A.

The second required condition for parallel resolution of mispredictions is that there should be no data dependence chains flowing to the second branch from regions that are control dependent on the first branch. Consider, for example the flow graph in Figure 2(b). The branch in block T is control independent of the branch in block Q but *data dependent* on the variable `i`, which is modified in blocks R and S, both of which are control dependent on the branch in block Q. The branch in block W, on the other hand, is both control independent of the branch at Q and *data independent* of blocks that are control dependent on the branch at Q. Following the notation introduced by Al-Zawawi et al [3] we use the acronyms *CIDI, CIDD,* and *CD* to indicate that an instruction X is, respectively, control independent and data independent of, control independent but data dependent on, or control dependent on a particular branch, B.

While control independence has been explored before in a variety of different circumstances, it is this second independence condition that we explore in this paper. In Sections 3 and 4 we show that a large fraction of the branches that postdominate low confidence branches are also data independent of those low confidence branches. We also show that as in the example in Figure 2(b), of the several branches that postdominate a low confidence branch, the ones that are "farther away" are often data independent even if the closest postdominating branch is data dependent.

2.3 Architectures that exploit BLP

Control Independence (CI) architectures are capable of executing mispredicted branches in parallel. Examples of CI architectures include Thread-Level Speculation or Speculative Multithreaded processors. These processors break



(a) A superscalar processor serializes branch mispredictions.

(b) Branch-mispredict level parallelism.

Figure 1: A timeline of two mispredicted branches (block identifiers refer to the flow graph in Figure 2(a). (a) The processor fetches block A, then (incorrectly) fetches block B. When the branch at A finally executes all the subsequently fetched instructions must be flushed. Even though block E was already fetched it needs to be fetched again. Thus the mispredicted branch at E is detected later than it could be. (b) If we knew that the branch at E could execute independently of the decision made by branch A we could start processing it sooner, overlapping the penalty paid by the two mispredicted branches.

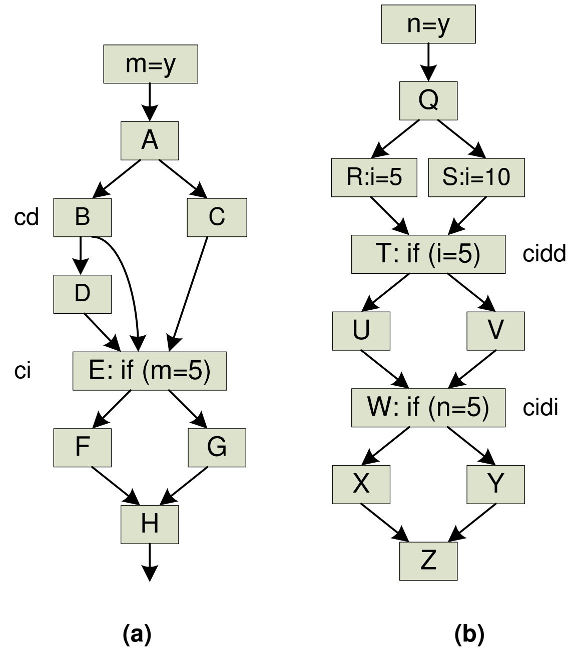


Figure 2: (a) Branch E is *control independent* (ci) of Branch A. Thus, Branch C does not necessarily need to be squashed and refetched if Branch A mispredicts. Branch B is *control dependent* (cd) on Branch A. Branch B is only usefully fetched on one of the two paths from Branch A. (b) Both Branch T and Branch W are control independent of Branch Q. Branch T, however, is *data dependent* (dd) on instructions that are control dependent on Branch Q, while branch W is *data independent* (di) of Branch Q. Thus Branch T can not be resolved before Branch Q is resolved, while branch W can.

up a single-threaded program into multiple threads that execute concurrently on an simultaneously multi-threaded (SMT) processor [26, 29, 2, 20, 1] or a multi-core machine [30, 12, 33, 22, 18]. Skipper [5] is similar to Speculative Multithreaded architectures implemented on an SMT processor, except that only one thread is allowed to fetch instructions at any given time.

Thread spawning in these CI architectures can be driven by a postdominator analysis [1, 7]. For example, in Figure 3, block E postdominates block A. Thus, whenever block A is reached, a new thread can be created starting at block E, since the processor is guaranteed to reach block E at some time in the future. For this thread, we call Block A the *spawner* block and call block E the *spawnee* block. The spawner thread fetches instructions as usual until it reaches block E. At this point, the spawner stops fetching instructions (the work it is about to begin has already been done by the spawnee thread). We say that the spawner thread *reconnects* with the spawnee. Since these processors spawn threads at control-independent points, branch mispredictions in one thread don’t affect instructions in other threads. These processors may thus have two mispredicted branches resolving in parallel, if the branches are in separate threads.

Although Speculative Multithreaded processors can exploit BLP, they are designed to exploit other program characteristics, like loop-level and data parallelism. Since BLP is an incidental phenomenon rather than the primary goal of these architectures, the amount of BLP exploited by these architectures can be low, such as for the naive CI architecture shown in Section 4.3. This paper shows that *through BLP-focused processor policies, a Speculative Multithreaded processor can exploit large amounts of BLP, which directly results in substantial performance gains.* Section 3 gives more details about these BLP-centric policies.

The CI architectures mentioned above fall in the general category of *proactive* CI architectures [13], where thread spawn is independent of branch mispredictions. On the other hand, *reactive* CI architectures [27, 3] reduce the impact of branch mispredictions by selectively re-executing only those instructions that are control-dependent on the

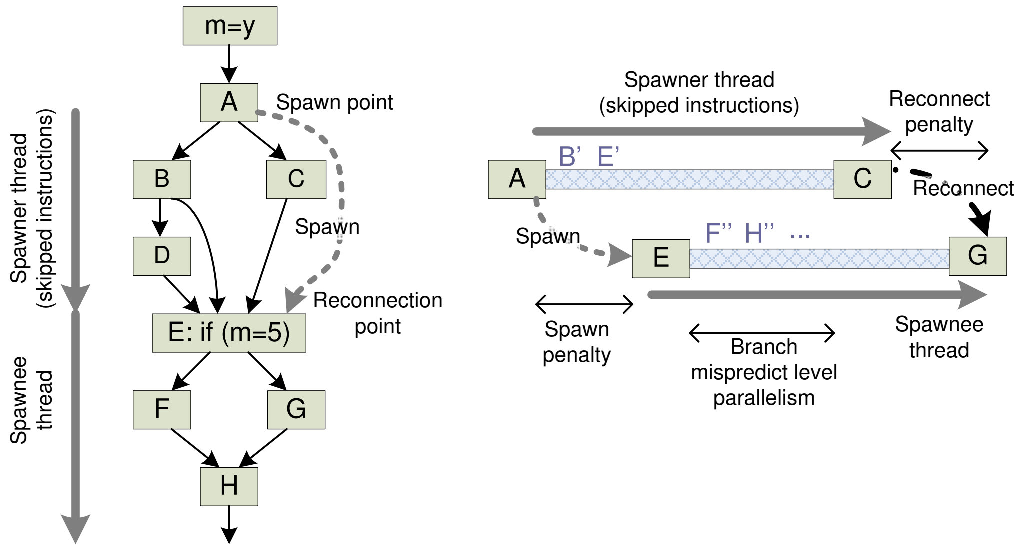


Figure 3: Terminology used in this section. The diagram on the left shows a static control flow graph. The diagram on the right shows a particular dynamic instantiation of the same code. In this case, the branch in block A is hard-to-predict, and the system has decided to create a new thread starting at the beginning of block E.

mispredicted branch. The focus of this paper is exploiting BLP in proactive CI architectures, although our techniques could also, potentially, improve the performance of reactive CI processors.

3 Strategies to Exploit BLP

The previous section introduced the concept of branch-mispredict level parallelism (BLP), and explained how CI architectures can leverage BLP to improve performance. In this section, we describe techniques that allow a CI architecture to maximize the BLP that it exploits in an application.

A CI architecture has a large number of spawn choices available to it. Section 3.1 shows, with an example from the SPEC2000 benchmark `vpr.route`, that spawn choice has a dramatic impact on the BLP exploited by a CI architecture. We also show that exploited BLP strongly correlates to improvements in performance.

Thus, a CI processor should choose spawns that maximize BLP. Towards this purpose, Section 3.2 describes profile-based estimation of the BLP that will be uncovered by different spawn choices. Section 3.3 presents examples for estimated BLP profiles in different benchmarks, and shows that estimates of BLP are strongly correlated with performance in full simulation. Finally, Section 3.4 shows that in addition to making the right spawn selection, a CI architecture also needs efficient dependence-handling mechanisms to maximize the exploited BLP.

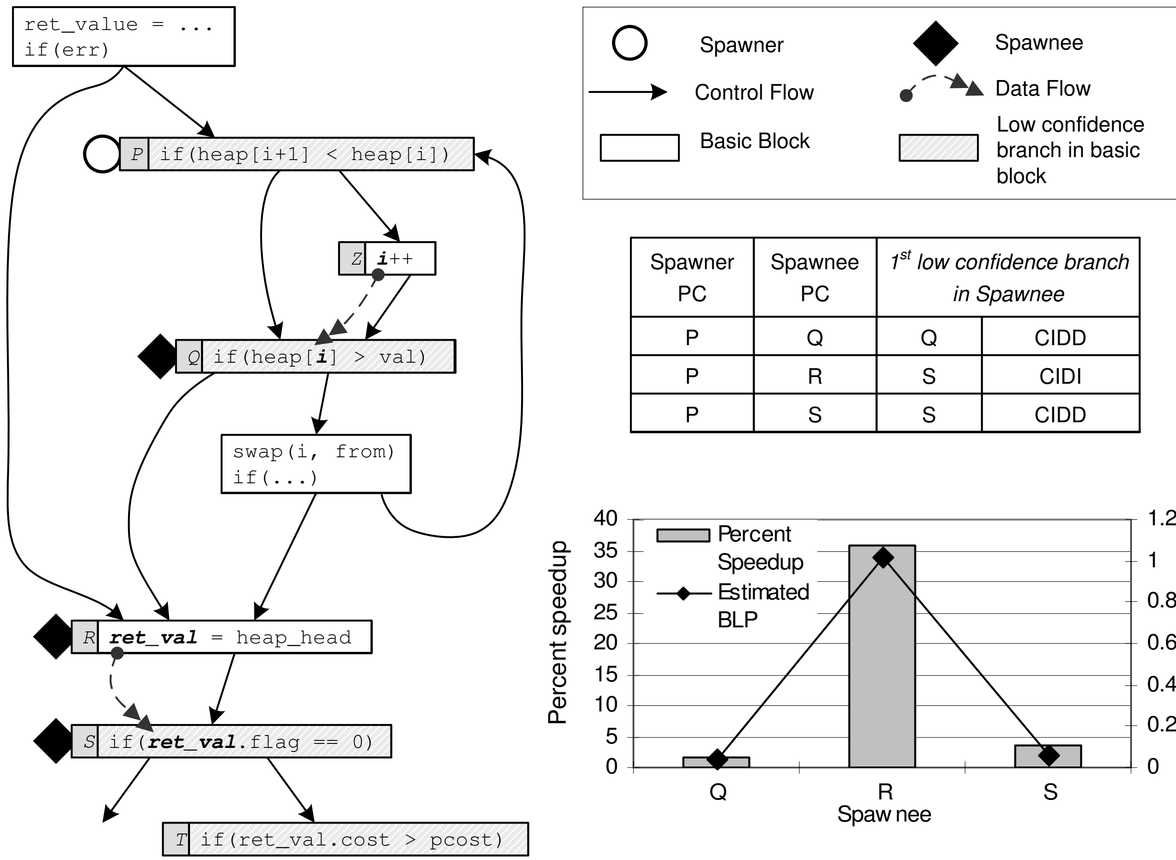
3.1 Making the Right Spawn Choice for BLP

This section presents an example illustrating the dramatic effect that spawn choices have on BLP. Figure 4 shows the control-flow graph for a loop from the SPEC2000 benchmark `vpr.route`. This loop has been inlined with its caller function, for ease of explanation. Each rectangle in the figure represents a basic block. Two important data-dependences are also shown. Branches that mispredict frequently are marked in gray.

P is a low-confidence branch (34% misprediction rate) that alone accounts for almost half of the dynamic mispredicts seen in `vpr.route`. As shown in Figure 4, P has 3 basic blocks (Q, R and S) as its postdominators. Thus, from P, a Speculative Multithreaded system has the choice to spawn either Q, R or S. These three potential spawnee points have different BLP characteristics.

Spawn Q: If Q is spawned, the first low-confidence branch that the spawnee thread will fetch is Q itself. However, because of the increment of variable i in block Z, branch Q is CIDD. This branch will not be executed until the spawner thread executes the increment instruction. As a result, even if branches P and Q both mispredict, we would not see an improvement in performance because the resolutions of P and Q are *serialized*.

Spawn R: On the other hand, if block R is spawned, S will be the first low-confidence branch (misprediction rate 39%) fetched by the spawnee thread. S is a CIDI branch, since it doesn't depend on any of the "skipped" instructions between block Q and block R. Moreover, following branch S, low-confidence branch T is also CIDI. Note that the spawner thread would be executing multiple instances of low-confidence branches P and Q in a loop. Thus, the two threads created by spawning block R from block P will both be executing a number of independent, low-confidence



Spawner PC	Spawnee PC	1st *low confidence branch in Spawnee*	
P	Q	Q	CIDD
P	R	S	CIDI
P	S	S	CIDD

Figure 4: The `getHeapHead()` function from `vpr.route`. The branch in basic block P is hard to predict (misprediction rate 34%). Blocks Q, R and S are the three postdominators of block P, and thus, are all control independent of P. Q is data dependent on P through the variable `i`. Block S is dependent on block R, thus if P spawns S directly the branch at S will be data *dependent*, but if P spawns R (which immediately falls through to S) then the branch at S will be data *independent*.

branches. In other words, this spawn choice exposes a large amount of BLP.

Spawn S: Spawning S directly from P will not lead to high BLP. Branches S and T depend on instruction R, and would therefore be marked CIDD in the spawnee thread. Note that the same branch (e.g, T) can be CIDI (when R is spawned) or CIDD (when S is spawned), depending on the choice of spawn point.

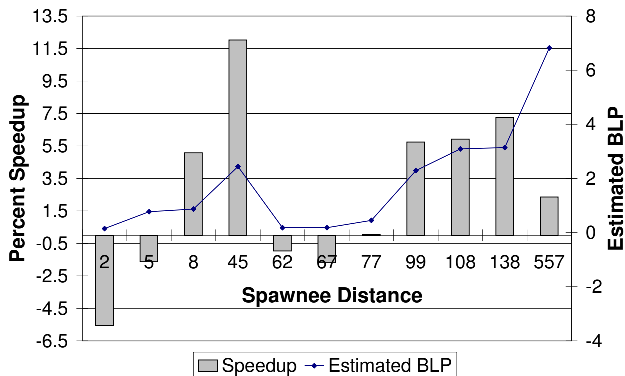
To investigate how these three spawn choices affect performance, we ran `vpr.route` on a 4 core Speculative Multithreading simulator, spawning only one of the three potential spawn points at a time. The machine configuration is described in Section 4.1. The graph in Figure 4 shows the results, where the y-axis on the left shows percent speedup over a superscalar. The y-axis on the right shows measured BLP, which is indicative of the extent to which mispredict penalties are overlapped.

Figure 4 shows that spawning block R is the superior choice: it results in a 35% speedup over a superscalar, while spawning Q or S results in hardly any speedup. It also demonstrates that spawning block R leads to a high degree of overlap of misprediction penalties, as indicated by the high value of measured BLP. Note that this performance difference arises even though spawnees R and S differ by only a single C statement.

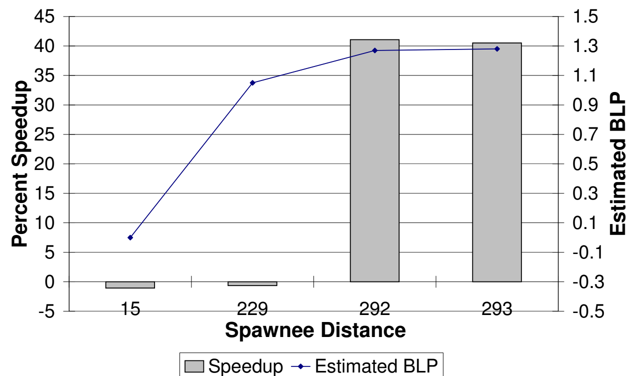
3.2 Estimating BLP for Spawn Choices

The previous section shows that spawn choice impacts BLP and thus performance. Our goal is to use BLP to drive an intelligent policy that chooses the best postdominator to spawn from a low-confidence branch. One possibility would be to build a microarchitectural mechanism that dynamically learns the BLP characteristics of different postdominators. In this paper, we use an alternative, profile-driven approach to choosing the best postdominator.

We have developed a profiler/trace-analyzer that chooses the best spawn point for low-confidence branches based on an estimate of BLP. We use strategies described in Sec-



(a) Estimated BLP metric for `twolf`.



(b) Estimated BLP metric for `vpr.place`.

Figure 5: Application speedup versus estimated BLP for different postdominators of particular hard-to-predict branches in `twolf` and `vpr.place`.

tion 4.4 to shortlist the possible set of candidate spawn pairs that could be useful. The analyzer next takes each possible pair one-by-one, and tries to estimate the BLP that would be exposed when the postdominator is spawned on a Speculative Multithreaded system. For each spawner, the spawnee with maximum estimated BLP is selected. Note that while estimated BLP and BLP are different measures, *estimated BLP* is strongly correlated with the BLP observed during full simulation.

In order to estimate BLP, we observe that a spawn will expose more BLP if the spawned thread fetches and resolves correct-path mispredicted branches *before* the spawner thread has finished executing. Note that we choose branches that mispredict frequently as spawners. Thus, if we spawn a postdominator that also fetches and resolves mispredicted branches, multiple mispredicted branches from the spawner and the spawnee thread are likely to be resolving in parallel. *Estimated BLP* is simply the average number of correct-path mispredicted branches resolved by the spawnee thread before the spawner thread has finished executing all its instructions. Note that CIDD mispredicted branches do not contribute to estimated BLP. Estimated BLP for a branch-postdominator pair can be found using profiling and memory-dependence analysis [23]. For the experiments in this paper, we use trace based dataflow-height analysis instead [19].

Going back to the example from `vpr.route`, we find that because of CIDD branches, spawning Q and S results in low estimated BLP (0.04 and 0.06 respectively). For example, if Q is spawned and the first instance of branch Q mispredicts, the spawnee thread will not be able to fetch any more correct-path instructions before the spawner thread reconnects to it. On the other hand, spawning R results in a estimated BLP of 1.02, because the spawned thread fetches CIDI low-confidence branches.

In the next section, we look at a few more examples from SPEC2000 benchmarks which indicate that estimated BLP is an excellent metric to choose spawns.

3.3 Estimated BLP and Performance

In this section, we investigate spawn choices for important low-confidence branches in two SPEC2000 benchmarks: `twolf` and `vpr.place`. We pick a single branch from each of these benchmarks that contributes a significant amount to the total number of branch mispredictions. We estimate BLP as described above, and observe performance using detailed simulation as described in Section 4.2. We find that the performance of a spawn point is strongly correlated to estimated BLP.

Figure 5(a) shows the performance and estimated BLP for different postdominators of a low confidence branch in the benchmark `twolf`.[1] The x-axis shows the average number of dynamic instructions on the path from the low-confidence branch to each of its postdominators. The left y-axis shows speedup over a superscalar processor, with machine parameters described in Section 4.1. The right y-axis shows estimated BLP. Recall that an estimated BLP of 2 indicates that on average, the spawnee thread is expected to fetch and execute two mispredicted branches before the spawner thread reconnects to it.

The structure of the code dictates the shape of the curves in Figure 5(a). This portion of `twolf` has a series of short loops, interspersed with if-else blocks. The low-performance postdominators in the beginning represent spawning to an if-else block, where the if-statement is CIDD. The low-performance postdominators around 65 dynamic instructions from the low-confidence branch are from a loop that uses values produced by the immediately preceding loop so these low-confidence branches are CIDD.

[1] All the experiments in this section, and with `vpr.route`, were performed with the register and memory dependence-based BLP optimizations enabled. These optimizations are described later.

The figure demonstrates that the estimated BLP for a spawn point is correlated with its performance in a detailed simulation. The last spawn point is an outlier that suffers from load imbalance problems for which the estimated BLP metric does not account.

Figure 5(b) shows the analogous graph for an important branch in `vpr.place`. We note that the second postdominator in `vpr.place` performs poorly, even though it has quite high estimated BLP. This is because of memory misspeculations, which are hard to estimate using a profile analysis.

These figures, and the `vpr.route` example from the previous section demonstrate that estimated BLP is a good metric for making a spawn selection. Note that while the immediate postdominator frequently gets low BLP, we are often able to find a better control-independent point to spawn to.

3.4 Improved dependence handling for BLP

Before a BLP-driven spawn selection policy is applied, we find that the BLP in the baseline configuration of CI architectures that spawn immediate postdominators of low-confidence branches can be quite low. Inefficient handling of register and memory dependences is a major cause of low BLP. With registers, the problem is false dependences introduced by restores to callee-saved registers. On the other hand, memory dependences are a problem because of squashes that are a result of true dependence violations between loads and stores in different threads.

Improving BLP across function calls

False register dependences because of restores to callee-saved registers reduce the amount of BLP that can be exploited across function calls. We describe this phenomenon in this section, and also present a simple hardware solution. To identify CIDD instructions, we use a register-dependence predictor that dynamically learns the set of registers written between the spawn PC and the reconnect PC. This predictor is quite similar to the one proposed by Cher [5].

The values of callee-saved registers are guaranteed to be preserved across function calls. If a function uses a callee-saved register, it must save the register on the stack, and restore it before returning to the caller. As has been observed by Ohsawa [25], if a callee-saved register is written between the spawn and the reconnect point, it does not necessarily represent a true data dependence: the write may simply be restoring the old value of the register. However, an unsophisticated register-dependence predictor would consider such writes as true data dependences. We find that false dependences arising from callee-saved registers reduce that amount of BLP that can be exploited across function calls, because more mispredicted branches in spawnee threads are marked CIDD. Such branches cannot execute in parallel with mispredicts in the spawner thread, thus decreasing BLP.

We extend Skipper's register synchronization mechanism to keep track of call-depth. Any writes to a callee-saved register that are performed at a call-depth greater than that of the spawn point do not train the predictor. Removing these false dependences improves performance by reducing the number of CIDD mispredicted branches, thereby increasing BLP. We analyze the performance of this optimization in Section 4.4. The effectiveness of this optimization is dictated by our instruction set architecture. Architectures, like SPARC, that use register windows can often avoid register spills during procedure linkage, while x86, with its small register space, tends to have more spills.

Memory Dependences

Most speculative multi-threaded systems suffer from the problem of inter-thread memory dependence violations. These violations occur when loads and stores to the same address are executed out-of-program-order in different threads. Such violations may flush already-resolved CIDI mispredicted branches, which decreases BLP because these branches will need to be fetched and executed again.

To reduce the number of memory violations, along with synchronizing registers, we also synchronize memory operations across different threads. We do this by treating store-set IDs [6] as architectural registers, and extending the Skipper register synchronization mechanism to keep track of the store-set IDs that are written between the spawn PC and the reconnection PC [21]. Thus, load instructions in a spawnee thread are occasionally marked as CIDD, if there is a store to the corresponding store-set in the skipped computation.

However, since store-sets are only approximate indicators of dependences between loads and stores, this technique of synchronizing loads and stores can sometimes impose conservative constraints, thereby decreasing performance. For example, the benchmark `vpr.route` operates on a large heap data structure, as shown in Figure 4. The loop in the figure performs stores to a large heap. As shown earlier, the best spawn, P-to-R, spawns across the entire loop. In the code below R (not shown in the figure), the heap is accessed again by loads that feed data to several low-confidence branches. These loads *occasionally* operate on the same addresses as the stores in the loop containing P, and thus, are assigned the same store-sets. However, since the heap is large, most of the time these loads don't access the heap entries written by the most recent execution of the loop. If load operations are synchronized conservatively, some loads will be marked CIDD unnecessarily. As a result, many low-confidence branches that were CIDI become CIDD instructions, thereby decreasing BLP.

To summarize, there is a tradeoff between decreasing the violation rate and increasing the number of CIDD branches. Instead of using an all-or-nothing approach to memory syn-

chronization, we use an adaptive policy that dynamically chooses whether or not to synchronize a load based on its violation rate. An adaptive memory-dependence predictor improves BLP and performance, by finding the right balance between reducing the violation rate and increasing CIDD mispredicted branches. We evaluate the performance of our memory synchronization mechanism in Section 4.4.

4 Experimental Results

We evaluate the performance of a CI architecture tuned to exploit BLP. We first describe the machine architecture and simulation infrastructure. Next, we investigate the performance of three BLP optimizations: removing false register dependences, adaptive memory synchronization, and a BLP-centric spawn selection policy.

4.1 Machine Architecture

We model a distributed architecture with 4 cores, where each core is a 4-wide, out-of-order superscalar processor. Important processor parameters are given in Table 1. Each core has local L1 instruction and data caches and branch predictors. The L1 data caches are kept coherent using an idealized, update-based protocol. In Speculative Multithreaded processors only one core is retiring instructions and committing stores to memory at a time and we model a single-cycle broadcast of retiring stores.

When a core (say P) fetches a specified spawner instruction it sends its neighbor (say Q) the program counter of the spawnee. We model a 4 cycle latency for this process. Register values that need to be communicated between cores are sent on a register-value bus with a latency of 4 cycles. On each core dependent instructions waiting for register or memory values from a predecessor thread to arrive (CIDD instructions) are stored in a FIFO called the *Divert Queue*, similar to the structure used by Al-Zawawi et al [3]. CIDI instructions are dispatched to the scheduler. Goodpath stores write their values to a single, chip-level speculative cache upon execution, as proposed by Garg et al [11]. About 4% of dynamic loads receive their data from this cache, which takes an extra 4 cycles. In the rare instance that a spawner store executes after a dependent spawnee load, an off-critical-path global load-queue detects the violation in the cycle after the store completes execution.

We use a mechanism similar to Skipper[5] to dynamically identify and handle inter-thread register dependences. As mentioned in Section 3.4, the same mechanism enforces inter-thread memory dependences by treating store set identifiers [6] as architectural registers [21]. Each core has a large, aggressive, tournament branch predictor (192KB). At large hardware budgets, this tournament predictor has an accuracy very similar to that of a perceptron predictor [14]. The branch predictors of all cores are trained by retiring branches. For a freshly-spawned thread, global history bits corresponding to the most recent branches (in program order) are not available: the instructions that generate these bits will be executed by the spawner thread sometime in the future. For most benchmarks, this phenomenon results in higher branch misprediction rates for the CI architecture compared to the superscalar, as shown in Table 2.

Parameter	Value
Pipeline Width	4 instrs/cycle (retire 8 instrs/cycle)
Branch Predictor	192KB Combined, 64KB gshare, 64KB bimodal, 64KB selector 18 bits of history
Misprediction Penalty	10 cycles
Reorder Buffer	512 entries
Scheduler	64 entries
Functional Units	4 identical general purpose units
L1 I-Cache	32Kbytes, 4-way set assoc., 128 byte lines, 10 cycle miss
L1 D-Cache	32Kbytes, 4-way set assoc., 64 byte lines, 10 cycle miss
L2 Cache	512Kbytes, 8-way set assoc., 128 byte lines, 100 cycle miss
Diverter Queue	128 entries
Spawn Latency	4 cycles
Inter-core Register Comm. Latency	4 cycles
Number of Store Sets	32
Register Dependence Predictor	2KB, 2-way assoc
Memory Dependence Predictor	2KB, 2-way assoc

Table 1: Pipeline parameters.

4.2 Simulation Methodology

Our 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 don't match. Whenever a branch misprediction is discovered, the simulator immediately reclaims backend resources (ROB and scheduler entries, etc.), restores the RAT from the branch's checkpoint, and begins fetching instructions from the correct target of the mispredicted branch.

For all the experiments in this paper, we execute 9 SPEC2000 integer benchmarks on the lgred input sets [17]. Our system libraries do not currently support `eon` and `gap`. We don't simulate `vortex` and `gzip` as these benchmarks have low mispredict rates (0.58% and 2.01% respectively) on the aggressive tournament predictor. The simulator skips the initialization phase of each benchmark, warming the caches, and then executes 100 million instructions. All the graphs that we present show the speedup of different Specu-

Benchmark	Superscalar	SpecMT
bzip2	9.30	10.64
crafty	5.20	6.11
gcc	4.71	4.99
mcf	4.45	4.56
parser	4.17	5.18
perlbmk	10.84	10.29
twolf	11.56	11.66
vpr.place	9.44	9.99
vpr.route	8.38	8.03

Table 2: Branch Misprediction Rates in Superscalar and SpecMT models

lative Multithreaded configurations over a superscalar with the same configuration as a single core of the Speculative Multithreaded processor.

Our spawn points are derived from control-independence and profile analyses. The control-independence analysis is performed on the benchmark binary and identifies the basic blocks that postdominate each branch. The profile analysis identifies high-BLP postdominators of low-confidence branches on the lgred trace. As proposed by Agarwal [1], a spawn cache can be used to store these postdominators. Since the total number of distinct static low-confidence branches is less than 100 for all applications other than `gcc`, which has 415, we don't model capacity or size constraints for the spawn cache.

4.3 Baseline Configuration

The baseline proactive control-independence configuration that we start with is called IPS (Immediate Postdominator Spawning). This configuration is similar to Skipper [5], except that we use a multi-core processor whereas the original Skipper proposal was restricted to exploiting control-independence on a single core. We use a 16KB enhanced JRS predictor to identify low-confidence branches. When a low-confidence branch is fetched and at least one of the four processor cores is idle, the processor spawns the low confidence branch's immediate postdominator as long as the postdominator is greater than 10 instructions and less than 500 dynamic instructions from the branch. The baseline IPS configuration does not perform any memory synchronization, nor does it remove false register dependencies arising from restores to callee-saved registers.

The leftmost bar in Figure 6 shows the relatively small performance improvements that our baseline IPS configuration attains over a superscalar, which is similar to the results in the original Skipper proposal [5], and recent research [13]. Cher and Vijaykumar attributed the modest performance gains seen by IPS architectures to low coverage of mispredicted branches [5]. While low coverage is certainly an issue, we find that CIDD mispredicted branches

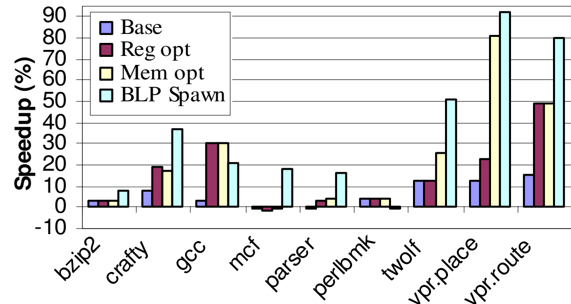


Figure 6: Speedup improvements from a BLP-centric spawn policy. The first bar represents spawning immediate postdominators of all low-confidence branches. Subsequent bars add register and memory-based BLP optimizations, and a BLP-aware spawn policy.

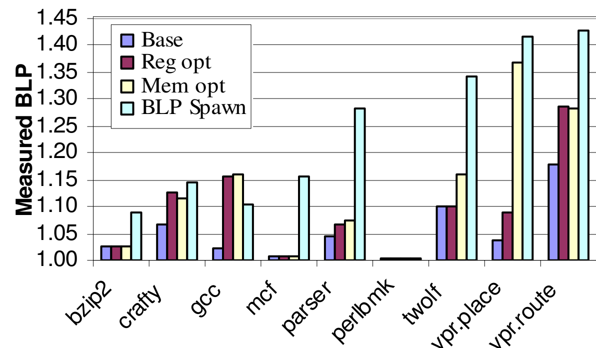


Figure 7: MeasuredBLP improvements from a BLP-aware spawn policy. Improvements in measuredBLP correlate strongly with improvements in performance.

are perhaps a bigger concern.

In particular, we find that the average dispatch-to-issue latency for mispredicted branches in the IPS processor is significantly higher than the corresponding latency in a superscalar. This effect is largely due to CIDD instructions that sometimes stay in the IPS processor's scheduler for *hundreds* of cycles, thereby mitigating the beneficial effects of spawning across low-confidence branches. As a result of the long latency of CIDD branches, the baseline IPS configuration is not able to exploit appreciable amounts of BLP, as shown in the leftmost bar in Figure 7.

4.4 BLP Optimizations

To improve the amount of BLP exposed by our Proactive Control-Independence architecture, we apply three BLP-improving optimizations: improving BLP across function calls by removing false register dependences, reducing memory violation squashes by adaptive memory dependence prediction, and choosing high-BLP spawns through profiling and trace-analysis.

Removing false register dependences

Section 3.4 pointed out that callee-saved registers introduce false register dependences which increase the number of mispredicted branches that are marked CIDD. The bar marked *Reg opt* in Figure 6 shows the performance improvement attained by removing these false dependences. Figure 7 shows the corresponding improvement in BLP. Removing these dependences leads to fewer CIDD branches, and improves BLP on `crafty`, `gcc`, `parser`, `vpr.place` and `vpr.route`. This BLP increase translates to performance improvement.

Adaptive memory synchronization

As indicated in Section 3.4, inter-thread memory violations reduce performance and BLP significantly. The bar labeled *Mem opt* shows the speedup attained by an adaptive memory dependence predictor that synchronizes loads if their violation rate is high, and otherwise speculates on loads [21]. The predictor extends Skipper's register synchronization mechanism to memory by treating store-sets as architectural registers. However, always synchronizing can be extremely conservative and increase the number of CIDD mispredicted branches. Adaptive memory synchronization strikes the right balance between synchronization and speculation. This leads to improvements in performance in benchmarks like `twolf` and `vpr.place`, where misspeculations were a problem, while preserving the benefits of load speculation.

BLP-based spawn selection

Finally, we investigate the performance effects of a profile driven spawn policy that uses BLP to choose which postdominator of a low-confidence branch should be spawned. In this configuration, we don't employ a dynamic confidence predictor to identify low-confidence branches. Instead, profiling is used to statically identify branches that mispredict frequently and contribute to 95% of the mispredicts. Their postdominators constitute the possible spawn points. To reduce the search space, we use profile information to discard postdominators that are too far in the future.

For every such branch, the postdominator that has the best *estimated BLP* is chosen. We often find low-confidence branches for which none of the postdominators have a high value of estimated BLP. Since low BLP values can cause slowdowns compared to a superscalar (because extra latency is added to CIDD mispredicts), an admittance threshold of 0.1 is used: any postdominator for which estimated BLP is less than the threshold is never spawned, even if it is the postdominator with the highest BLP. Although this decreases the overall coverage of mispredicted branches, we find that it results in better performance.

The bar labeled *BLP Spawn* in Figure 6 shows the speedup of a BLP-centric spawn policy, over a policy that spawns only the immediate postdominator (IPS). As can be seen, BLP-centric spawning improves the performance of all applications significantly, except for `gcc` and `perlbmk`. Figure 7 shows the corresponding BLP. The increases in BLP are correlated with increases in performance.

For gcc, performance decreases slightly because of our static BLP analyzer chooses branches that account for only 80% of the mispredicts in gcc (as opposed to 95% for other applications. This is because for 95% coverage of mispredicts, the analyzer would need to consider 50,000 branch-postdom pairs, which it is not equipped to handle. We analyze the performance of `perlbmk` and `bzip2` in the next section.

Benchmark Analysis

Perlbmk is one benchmark which does not benefit from control-independence, even though it has a high mispredict rate. It also shows no branch-mispredict level parallelism. This is because more than 90% of the mispredicts in perlbmk can be attributed to a single, indirect call instruction. Perlbmk is an interpreter program. It has a number of processing functions, one for each opcode that it expects to interpret. In the main loop, a function pointer is used to decide which processing function should be called. This indirect call (in function `runopsStandard`) mispredicts because it gets an incorrect target from the branch-target buffer (BTB). The next opcode is read at the end of the processing function, and returned back to the caller. The main loop of perlbmk is shown below:

```
while (PLop = ((*PLop)->opPpaddr)(ARGS));
```

Intrinsically, there is little BLP in this code. If the next loop iteration is spawned when the indirect call is reached, the spawnee thread will get stuck at a CIDD mispredicted branch (the next indirect call). This branch is CIDD, since the current processing function returns the next function that should be called. Thus, neither the IPS configuration, nor a BLP-centric configuration achieves significant performance gains on perlbmk.

As far as bzip2 is concerned, it spends a large amount of time in sorting routines. For arrays that are short (encountered when bzip2 starts compression or decompression), bzip2 uses a sorting function called `simpleSort`. The sorting loop in this function can be spawned across, and relatively high speedups (80%) are observed. However, for a large part of its execution, bzip2 encounters longer arrays which are sorted using a version of quicksort. Quicksort is known to be a highly sequential algorithm, and we are unable to discover high-BLP spawns during this phase.

5 Conclusions and Future Work

CI architectures can search for useful work to do in the shadow of a mispredicted branch. In this paper we have argued that the most useful work to do in the shadow of a

mispredicted branch is to resolve other branch mispredictions. We introduced BLP, which is a metric that quantifies the extent to which mispredict penalties are overlapped.

Although CI architectures are capable of executing mispredicts in parallel, we found that the amount of incidental BLP exploited by these architectures tends to be quite small. However, by focussing processor policies on mispredict parallelization, we can dramatically improve the BLP exploited by a CI architecture. Efficient dependence-handling mechanisms are required to expose more BLP. More importantly, we find that the spawns chosen by a CI architecture have a strong impact on the amount BLP that is extracted. We have demonstrated that the choice of where to start a spawned thread has dramatic consequences on BLP, and, as a result, performance. Choosing a control independent spawn point that is just a few instructions earlier or later can change the data dependence structure, changing the next mispredicted branch from CIDI to a CIDD or from CIDD to CIDI. When the next mispredicted branch is CIDI rather than CIDD we find more parallelism. In our ongoing work we are improving our data dependence analyzer in two ways. First we are improving the efficiency of the offline algorithms, and second we are working on microarchitectural mechanisms to dynamically search for spawns with better BLP characteristics.

While branch mispredicts have a first order impact on performance of integer applications, other performance degrading events like cache-misses may be more important for some applications. We are working on extending the proposed spawn selection techniques to extract more memory-level parallelism (MLP) in Speculative Multithreaded processors.

Acknowledgments

We thank Sanjay Patel and Steve Lumetta for valuable feedback on early formulations of this work, and Nitin Navale for contributions to the infrastructure. The work reported in this paper was supported in part by the National Science Foundation under grant CCR-0429711 and by the Gigascale Systems Research Center, one of five research centers funded under the Semiconductor Research Corporation's Focus Center Research Program. Computational resources were provided by the Trusted ILLIAC Center at the Information Trust Institute and Coordinated Science Laboratory at the University of Illinois. Sam Stone is supported by a National Science Foundation Graduate Research Fellowship. 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.

Bibliography

[1] M. Agarwal, K. Malik, K. M. Woley, S. S. Stone, and M. I. Frank. Exploiting postdominance for speculative parallelization. *Int'l Symp. High Performance Comp. Arch.*, (HPCA-13):295–305, 2007.

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

[3] A. S. Al-Zawawi, V. K. Reddy, E. Rotenberg, and H. H. Akkary. Transparent control independence (TCI). *Int'l Symp Comp Arch*, (ISCA-34):448–459, 2007.

[4] E. Borch, S. Manne, J. Emer, and E. Tune. Loose loops sink chips. *Int'l. Symp. High Performance Comp. Arch.*, (HPCA-8):299–310, 2002.

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

[6] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. *Int'l Symp Comp Arch*, (ISCA-25):142–153, 1998.

[7] P. K. Dubey, K. O'Brien, K. M. O'Brien, and C. Barton. Single-program speculative multithreading (SPSM) architecture: compiler-assisted fine-grained multithreading. *Conf on Parallel Arch and Compilation Techniques*, (PACT-1):109–121, 1995.

[8] 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.

[9] B. Fields, S. Rubin, and R. Bodík. Focusing processor policies via critical-path prediction. *Int'l Symp Computer Architecture*, (ISCA-28):74–85, 2001.

[10] M. Franklin and G. S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelism. *Int'l Symp Comp Arch*, (ISCA-19):58–67, 1992.

[11] A. Garg, M. W. Rashid, and M. Huang. Slackened memory dependence enforcement: Combining opportunistic forwarding with decoupled verification. *Int'l Symp Comp Arch*, (ISCA-33):142–154, 2006.

[12] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2), 2000.

[13] A. Hilton and A. Roth. Ginger: Control independence using tag rewriting. *Int'l Symp Comp Arch*, (ISCA-34):436–447, 2007.

[14] D. A. Jimeñez and C. Lin. Dynamic branch prediction with perceptrons. *Int'l Symp High Performance Computer Arch*, (HPCA-7):197–206, Jan. 2001.

[15] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. *Int'l. Symp. Computer Architecture*, (ISCA-31):338–349, 2004.

[16] H. Kim, J. A. Joao, O. Mutlu, and Y. N. Patt. Diverge-Merge Processor (DMP): Dynamic predicated execution of complex control-flow graphs based on frequently executed paths. *Int'l Symp. Microarchitecture*, (MICRO-39):53–64, 2006.

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

[18] V. Krishnan and J. Torrellas. A chip multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 47(9), September 1999.

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

[20] 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.

[21] K. Malik, M. Agarwal, and M. I. Frank. Adaptive memory synchronization (AMS): Balancing the risks and benefits of inter-thread load speculation. *Reconfigurable and Adaptive Architecture Workshop*, (RAAW-2), 2008.

[22] P. Marcuello, A. González, and J. Tubella. Speculative multithreaded processors. *Int'l. Conf. Supercomputing*, (ICS-12):77–84, 1998.

[23] M. Mock, M. Das, C. Chambers, and S. J. Eggers. Dynamic points-to sets: A comparison with static analyses and potential applications in program understanding and optimization. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, pages 66–72, June 2001.

[24] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. *Int'l Symp High Perf Comp Arch*, (HPCA-9):129–140, 2003.

[25] T. Ohsawa, M. Takagi, S. Kawahara, and S. Matsushita. Pinot: Speculative multi-threading processor architecture exploiting parallelism over a wide range of granularities. *Int'l Symp Microarchitecture*, (MICRO-38):81–92, 2005.

[26] I. Park, B. Falsafi, and T. N. Vijaykumar. Implicitly-multithreaded processors. *Int'l. Symp. Comp. Arch.*, (ISCA-30):39–51, 2003.

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

[28] E. Rotenberg and J. E. Smith. Control independence in trace processors. *Int'l. Symp. Microarchitecture*, (MICRO-32):4–15, 1999.

[29] A. Roth and G. S. Sohi. Speculative data-driven multithreading. *High Perf. Computer Arch.*, (HPCA-7):37–48, 2001.

[30] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. *Int'l Symp Computer Architecture*, (ISCA-22):414–425, 1995.

[31] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. *Int'l. Symp. Computer Architecture*, (ISCA-29):25–34, 2002.

[32] 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.

[33] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. *Int'l Symp Computer Architecture*, (ISCA-27):1–24, 2000.

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