Fetch-Criticality Reduction Through Control Independence

Mayank Agarwal, Nitin Navale*, Kshitiz Malik, Matthew I. Frank
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
{magarwa2,navale,kmalik1,mif}@uiuc.edu

Abstract

Architectures that exploit control independence (CI) promise to remove in-order fetch bottlenecks, like branch mispredicts, instruction-cache misses and fetch unit stalls, from the critical path of single-threaded execution. By exposing more fetch options, however, CI architectures also expose more performance tradeoffs. These tradeoffs make it hard to design policies that deliver good performance.

This paper presents a criticality-based model for reasoning about CI architectures, and uses that model to describe the tradeoffs between gains from control independence versus increased costs of honoring data dependences. The model is then used to derive the design of a criticality-aware task selection policy that strikes the right balance between fetch-criticality and execute-criticality. Finally, the paper validates the model by attacking branch-misprediction induced fetch-criticality through the above derived spawn policy. This leads to as high as 100% improvements in performance, and in the region of 40% or more improvements for four of the benchmarks where this is the main problem. Criticality analysis shows that this improvement arises due to reduced fetch-criticality.

1 Introduction

Superscalar performance is limited by the restriction of in-order fetch. The majority of instructions on the program critical path are fetch critical [9]. Events like branch mispredicts and instruction cache misses delay the fetch of future instructions.

Architectures can exploit the control-independence property of applications to attack the fetch-criticality problem. Control independence dictates that irrespective of the direction in which control flows for a branch, it is guaranteed to reconverge at postdominators, or control-independent points of that branch. Superscalar processors delay the fetch of control-independent points due to fetch serialization. On the other hand, control-independence architectures can concurrently fetch instructions that are control independent of fetch lateness-generating events such as branch mispredicts, thereby alleviating the fetch-criticality bottleneck.

Control-independent fetch, however, creates its own challenges arising from data-dependences between control-independent regions. These data-dependences make it hard to achieve the potential of control independence and to reason about strategies for improving performance. Policies and mechanisms that are naive in handling these risks can, in fact, degrade performance.

This paper presents a model for a control-independence (CI) based parallelization architecture. The model is an extension of the superscalar criticality model of Fields, Rubin and Bodík [9]. Reasoning in this model, we describe how control-independent spawning can alleviate the fetch-criticality bottleneck in the superscalar fetch mechanism, and explain the trade-off between fetch-criticality and execute criticality that governs the success of CI parallelization techniques.

Using this model, we then derive rules for successful spawning in terms of program criticality. These rules lend themselves to an implementable criticality-aware spawn selection policy. We describe how to extend the Fields' dynamic token passing mechanism to collect information needed by the policy.

Finally, we validate the model by attacking the problem of branch-mispredict induced fetch-criticality in superscalar execution. Detailed evaluation of the criticality-aware spawn policy show that it is robust and that it improves performance by reducing fetch-criticality.

The rest of the paper is structured as follows. In Section 2, we motivate the fetch-criticality bottleneck of superscalar processors, how control independence can be used to alleviate this bottleneck, and the trade-offs in this parallelization technique. Section 3 describes a spawn policy that can make spawn selections that break critical fetch-dependences without aggravating data-dependences. Section 4 describes how to extend Fields' token passing predictor for our environment. Evaluations in Section 5 confirm our insight. Finally we describe related work in Section 6, and draw conclusions and future directions in Section 7.

*Now at AMD Corp.

2 Fetch Criticality and Control Independence

Lateness of fetch is a major bottleneck to superscalar performance. Section 2.1 gives an overview of Fields superscalar criticality model. Instructions that are fetched late, are referred to as *fetch-critical* instructions in this model. In Section 2.2, we focus on the *fetch criticality generating events (FCGEs)* that delay fetch. Section 2.3 introduces control-independence (CI) architectures. In Section 2.4, we extend the criticality model of Fields et al for CI architectures and show how control-independent spawning can profitably remove previously fetch-critical instructions from the critical path. Section 2.5 describes the risk of spawning, in terms of the trade-off between fetch- and execute-criticality.

2.1 A Criticality Model for Superscalars

The Fields, Rubin, Bodík (FRB) criticality model is based on a dependence graph [9] that models the execution of instructions in a superscalar processor. The criticality graph is a directed graph induced on the trace of committed program instructions. As shown in Figure 1, each instruction is represented by three nodes. The first node (labeled "D") represents, in addition to *dispatch* of the instruction, its address generation, fetch, decode and renaming. The "E" node represents (out-of-order) issue and *execution* of the instruction. The "C" node represents instruction *commit*.

Graph edges represent dependences. True dataflow dependences are captured through EE edges, from the E node of producer to those of consumer instructions. Further, several edges model microarchitectural constraints. For an instruction, dispatch precedes execution, which in turn happens before commit. Thus, within each instruction, there is a DE edge, and an EC edge. Additionally, in a superscalar processor, all instructions are fetched and dispatched in-order, so a DD edge flows between successive instructions. Likewise, in-order retirement of instructions leads to a CC edge from an instruction to the subsequent instruction. The processor's reorder buffer contains only N instructions so the processor must stall the fetch unit whenever there are more than N uncommitted instructions. Thus there is a CD edge from each instruction to the Nth succeeding instruction in the trace.

Several additional edges represent dynamically occurring events. The execution of a mispredicted branch causes the machine to roll back state, and restart fetching from the correct target. This is represented by an ED edge from the E node of the mispredicted branch to the D node of the succeeding instruction, representing the correct target. The trace contains only instructions that are eventually committed, so the incorrectly fetched instructions are not included in the trace.

The Program Critical Path: Each graph edge is labeled with the latency induced by the dependence. For example, EE edges are labeled with the instruction's latency through issue contention and functional unit latency. Given a set of edge labels, we assign a time to each node. This is done using Wall's efficient algorithm for trace-based microarchitectural simulation [39]. For each node we look at all its incoming edges, and calculate a time by taking the time associated with the producer node and adding the assigned edge weight. The time associated with a node is the maximum of all the times calculated for its incoming edges. This represents the idea that each node of each instruction may not start its action until all of its dependences are satisfied. These dependences decide the program running time. In particular, *the longest path from the Dispatch node of the first instruction to the Commit node of the last instruction represents the* **critical path** *of the program [9].*

We say that an instruction is *critical* if any of its three nodes is on the critical path through the program. We say an instruction is *fetch-critical* if its dispatch node is on the critical path. An instruction is *execute-critical* if its execute node is on the critical path (but not dispatch node). We call an instruction *commit-critical* if only its commit node is on the critical path.

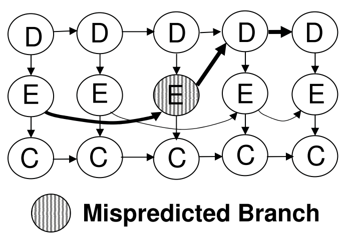
Dynamic Prediction of Criticality: Fields et al also showed how to construct an on-line predictor to dynamically learn and use information about instruction criticality. The predictor, which uses the *token-passing* mechanism, leverages the insight that if a dynamic instruction i is not on the program critical path, it will satisfy one of the following two conditions:

- None of the edges going out of instruction i is the last edge to arrive at any of the target nodes.
- The instructions nodes where i's outgoing edges are last-arriving are themselves not critical.

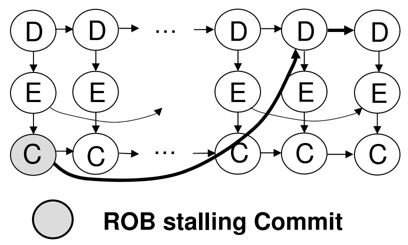
Briefly, the *token-passing detector* of criticality "plants" a token at the instruction whose criticality is to be assessed, and observes if the token propagates along a chain of last-arriving edges (i.e. it doesn't violate any of the above conditions) for a long time. If so, the instruction is deemed to be critical (or near-critical). Further details of the mechanism are described in [9].

2.2 Fetch Criticality Generating Events

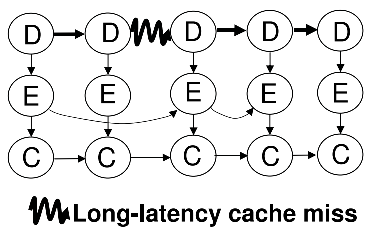
In a 4-wide superscalar, we find that on an average across the SPEC 2000 integer benchmarks, about 52% of all instructions are critical. About 66% of the critical instructions are fetch critical with the other 34% divided between execute- and commit-critical instructions. Fields et al showed that wider superscalars become even more fetch criticality constrained [9].



(a) Branch Misprediction: The correct target instruction of a mispredicted branch cannot be fetched until the branch instruction is executed to detect the incorrect prediction.



(b) Fetch Unit Stall: When the reorder buffer or scheduler is filled to capacity the fetch unit must be stalled. This introduces the possibility of making the first stalled instruction fetch-critical.



(c) Instruction Miss: Instruction cache misses extend the length of the path through the dispatch node of one instruction, possibly making subsequent instructions fetch-critical.

Figure 1: Branch mispredicts, fetch unit stalls and instruction cache misses are fetch criticality generating events (FCGEs).

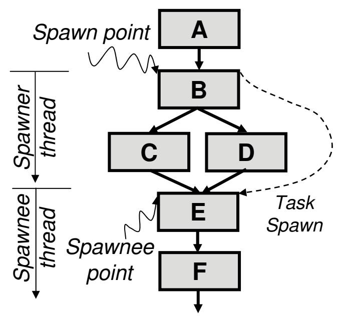


Figure 2: Terminology to describe the actions taken by our control-independence processor to find parallelism.

We can use the criticality model to focus on the instructions that delay the fetch of later instructions. We call these instructions *fetch criticality generating events (FCGEs)*. Fetch criticality generating events fall, roughly, into three categories. The first is branch mispredicts, which delay the fetch of the correct target of the branch instruction. The second is long-latency instructions that cause the scheduler or reorder buffer to fill, stalling the fetch unit. The third is instruction cache misses, which directly delay fetch of an instruction. Figure 1 summarizes the three types of FCGEs.

FCGEs add latency to the dispatch node of the first future instruction whose dispatch is affected. This can result in long chains of last-arriving edges along the dispatch nodes of future instructions. In the next section, we describe how control-independence architectures can reduce fetch-criticality by breaking such fetch-critical chains.

2.3 Control-Independence Architectures to Alleviate Fetch Criticality

In a program flow graph we say that one instruction X *postdominates* another instruction A iff all paths through the flow graph from A to the exit pass through X [7]. In other words, X postdominates A if X is guaranteed to execute after A executes, regardless of intervening control decisions. As an example, in the flow graph of Figure 2, block E postdominates block A.

Control-independence architectures exploit the postdominance property to find useful work to do in the shadow of mispredicted branches [1, 3, 5, 13, 27]. Thread-level Speculative and Speculative Multi-threaded processors find parallelism in a single thread of execution by breaking the thread into multiple threads that are executed concurrently [2, 12, 17, 18, 21, 24, 29, 33, 35]. The control-independence property can also drive the choices of spawns in these machines [1]. For example, in Figure 2 a new thread could be created at block E whenever block B is reached, since the processor is guaranteed to reach block E at some time in the future.

In this case, we would call the new thread generated starting at block E the *spawnee* thread with E being the spawnee block, and B the *spawner* block. The spawner and spawnee threads can concurrently fetch instructions. When the spawner thread reaches block E, it stops fetching instructions (the work it is about to begin has already been done by the spawnee thread). At this point, the spawner thread *reconnects* with the spawnee.

Instructions in the spawnee thread that depend on data produced in spawner thread can be handled in a variety of ways. They can be speculatively executed assuming that the data is available[2, 17], and signal a misspeculation if a violation is later detected. Or a data-dependence predictor [22] can be used to identify such instructions, which can then be delayed (synchronized), until data value is (conservatively) released by the spawner thread.

Since these architectures spawn threads at control-independent points, FCGEs in one thread need not affect instructions in other threads. Stated another way, branch mispredicts and other FCGEs in the spawner thread need not delay the fetch of instructions in the spawnee. This represents an opportunity to alleviate fetch criticality of delayed instructions. The next section presents a model that allows us to reason more formally about these architectures.

2.4 A Criticality Model for CI Architectures

In this section, we extend the critical path model described in Section 2.1 for control-independence architectures. Control-independence architectures remove the restriction of in-order fetch. On the other hand, delay might be added to inter-thread dataflow. Exactly how much delay is added depends upon the particular data-dependence handling technique used. Further, techniques such as data speculation can lead to thread squashes that need to be modeled.

Table 1 lists the edges that need to be added to the original criticality model of Fields et al, in order to model these effects. The first modification is a new DD edge going from the dispatch node of the *spawner* to the dispatch node of the *spawnee*. Note that there is no longer a dispatch edge from the last instruction in a spawner thread to the first instruction in the next thread, which is another modification from the superscalar model. This means that the spawnee can start dispatching as soon as the spawner dispatches (after some penalty, equal to the latency on the particular DD edge, which can be used to model a *spawn penalty*). Intra-thread dispatch proceeds as before.

The second modification is that CD edges impact dispatch only within a thread, and not across threads. This means that back end stalls in one thread need not stall dispatch in the successor threads, effectively allowing for a distributed window of instructions. In addition there is another CD edge to model finite thread resources.

Thirdly, the Fields' model already contains an ED edge to model branch mispredicts. We generalize the notion of mispredicts to capture intra-thread store-load violations, as well as inter-thread violations due to failed data speculation. Frequent data misspeculation can also be a FCGE, and can be treated in a similar manner as branch mispredicts. Note that the latency for detecting and communicating the failed speculation is implicitly captured through the weight given to this edge.

Finally, depending on the specific policy for handling inter-thread data dependences for a given control-independence architecture, delay might be added along EE edges that cross thread boundaries. This can model, for instance, the latency of inter-core communication, as well as the delay for architectures that synchronize on data dependences.

2.5 The Cost of Control Independence

By removing the restriction of in-order fetch/dispatch, control-independent spawning can make the previously fetch-critical CI points of FCGEs non-critical. **However, there is a cost to spawning**. The cost arises from the added delay to EE edges crossing thread boundaries. Breaking a fetch criticality chain by spawning can cause the new program critical path to flow through one of the other previously *near-critical* paths. If a delayed EE edge happens to be on one such path, where the slack [8] on that path is less than the delay added to the EE edge, then spawning was a bad idea since the program critical path was made worse by spawning.

Aggressive data dependence handling techniques, that speculatively execute instructions in spawnee assuming no dependence, try to shift the balance in favor of spawning by delaying as few inter-task edges as possible. However, they introduce the cost of misspeculations, which can become an important FCGE by significantly delaying fetch beyond a misspeculating consumer in the spawnee. Conservative techniques that synchronize likely consumers in the spawnee try to delay *likely dependences* until they are safe to be released. However, such techniques delay the communication of data values as well as run the risk of unnecessarily delaying a large number of instructions in the spawnee.

Regardless of the specific policy used, data-dependences (EE edges) that cross task boundaries introduce a cost to spawning. Thus, while spawning can alleviate fetch-criticality, it can make execute-criticality of such nodes (with inter-task EE edges) in the spawnee significantly worse, resulting in an overall loss in performance. Spawn selection policy is the deciding factor in this trade-off space. For any spawner, there is a large choice of spawnees, obtained from the post-dominance analysis. Section 3 shows how to select spawns that strike the right balance between fetch- and execute-criticality.

3 Criticality and Spawn Selection

Section 3.1 describes two rules for making good spawn choices, and eliminating bad spawns. These rules will be later used in the design of a spawn selection policy. The rules are illustrated through an example in Section 3.2. Finally, in Section 3.3 we elaborate on how program dataflow behavior impacts spawn profitability.

3.1 Rules for Spawn Success

The conditions for a successful spawn can be summarized by two simple rules:

1. The spawnee should be a fetch-critical instruction.
2. The slack on the EE edges that cross the task boundary as a result of the task spawn must be greater than the data latency added by the spawn mechanism.

As described in Section 2, control-independent spawning can be profitable by alleviating fetch-criticality in applications. Rule 1 states that if the spawnee point is not fetch-critical in the first place, then the spawn is largely useless since it tries to address a problem that does not exist (it

Name	Constraint modeled	Edge	Comment
DD	In-order dispatch (**non-spawn**)	$D_{i-1} \rightarrow D_i$	Non-spawnee instruction i cannot dispatch before $i-1$.
DD	**In-order dispatch (spawn)**	$D_{i-s} \rightarrow D_i$	Spawnee instr i cannot dispatch before spawner $i-s$ (s = spawn distance).
CD	Finite reorder buffer size	$C_{i-r} \rightarrow D_i$	**Instr $i-r$ and instr i are in the same thread** and r is the size of the reorder buffer
CD	**Finite thread resources**	$C_{last(i-n)} \rightarrow D_{first(i)}$	Instr $last(i-n)$ is the last instr in thread i-n, and instr $first(i)$ is the first in thread i and n is the number of thread contexts
ED	Failed Speculation	$E_{i-1} \rightarrow D_i$	Instr $i-1$ is a misspeculating instruction (mispredicting branch/load, etc), **and i is in same thread.**
DE	Execution follows dispatch	$D_i \rightarrow E_i$	An instr cannot execute before it has dispatched.
EE	Data dependences	$E_j \rightarrow E_i$	Instr j produces an operand of i. **Inter-thread data flow can have longer latency than intra-thread data flow.**
EC	Retire follows execution	$E_i \rightarrow C_i$	An instr cannot retire before execution.
CC	In-order retirement	$C_{i-1} \rightarrow C_i$	Instr i cannot retire before $i-1$.

Table 1: FRB criticality model dependence rules [9] adjusted for control-independence processor (**bold**).

could, however, create new problems by delaying EE edges that don't have enough slack).

Rule 2 states that it is not enough to break fetch-criticality chains, the objective of spawning should be to speed up the program execution. This means that the new program critical path should not be worse than the original path (that passed through dispatch node of the spawnee). In particular, the only reason why it could be worse off than the original path, could be if it flows through one of the EE edges that are delayed due to the act of spawning.

3.2 Spawn Rules in Action

In this section, we illustrate the above rules through a detailed example from the SPECInt2000 benchmark *twolf*. Figure 3 shows the control-flow graph of a fragment of interest in the function *new_dbox_a*, which accounts for a large fraction of the overall execution time. The node marked A is a branch that is likely to mispredict and generate fetch-criticality. We find that its postdominators: C, D, E, F, and G, are all likely to be fetch-critical (note that blocks D and F also contain FCGEs). In general we find that postdominators of blocks containing low-confidence branches have a high likelihood of being fetch-critical. Thus a dynamic instance of any of of these postdominators, when spawned from A, is likely to satisfy Rule 1 for spawn success.

On the other hand, not all of these postdominators are likely to satisfy Rule 2. Figure 3 also shows dataflow edges that are likely to be on the program critical path. In particular, the statement that produces *rowsptr* in block C is on the backward slice of likely-to-mispredict branches in blocks D and F. Thus, these EE edges from C to D and C to F are likely to have very little slack, since almost all mispredicted branches are on the program critical path in a superscalar processor. Spawns which add delay to these edges are likely to violate Rule 2.

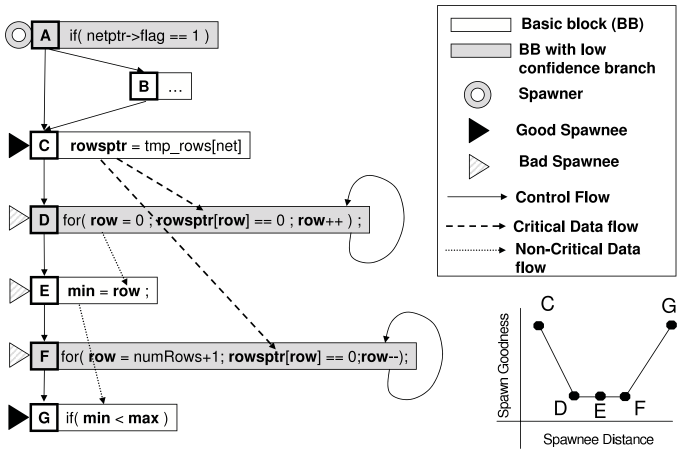
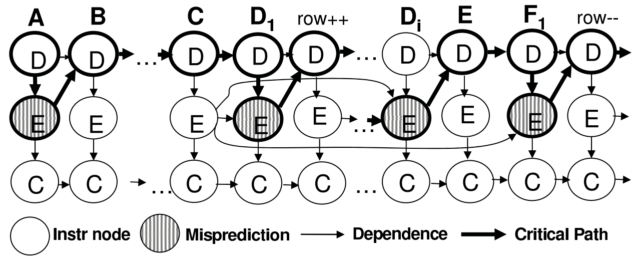
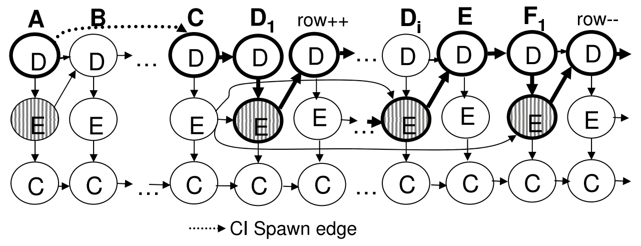


Figure 3: An interesting example from the `new_dbox_a` subroutine in twolf (somewhat simplified), showing spawnee choices for low confidence branch A. Spawnees D, E and F introduce inter-task data-dependence edges (through rowsptr) leading into already execute-critical nodes, and are bad. C and G are good spawns, since the subsequent execute-critical instructions are data-independent.

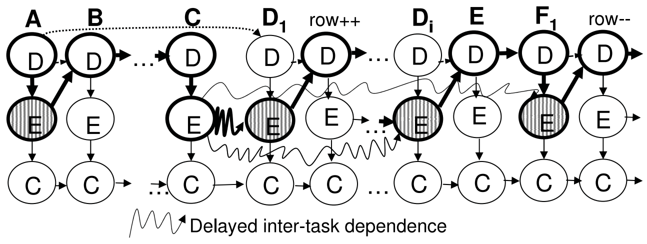
In this case, we find that when we spawn D, E, or F from A, one (or both) of the EE edges flowing the value



(a) Superscalar execution: Branch mispredicts make a large number of instructions fetch-critical. Note that the flow of *rowsptr* from C to D is near-critical.



(b) Good Spawn: CI spawn of C from A allows fetching C faster than earlier. It also removes execution of A (mispredict) from critical path.



(c) Bad Spawn: Even though spawning D fetches it faster, it adds considerable delay to a near-critical EE edge flowing *rowsptr*, which now crosses task boundaries. As a result, performance is degraded.

Figure 4: Critical path for superscalar execution and different spawn scenarios.

of *rowsptr* out of C gets delayed. Therefore, we can rule out these spawns as bad spawns. On the other hand, if we spawn C or G, we find that the EE edges that cross the task boundaries have a lot of slack. In particular, when we spawn G, the edge corresponding to the value *min* that flows from E to G has a lot of slack since the branch in G is usually predicted correctly. So even though the $E \rightarrow G$ EE edge is delayed, the delayed execute node of G is unlikely to be on the program critical path.

This is further explained through a set of criticality diagrams. Figure 4(a) shows the critical path in a normal superscalar execution of the program. Branch mispredicts of A, D, and F are on the program critical path. CI spawning is an attractive proposition to reduce the impact of misprediction of A on fetch-criticality of its control-independent instructions. In order to achieve this, we can spawn one of the CI points of A, which are likely to be fetch-critical (thus satisfying Rule 1). The available spawnee choices are: C, D, E, F and G.

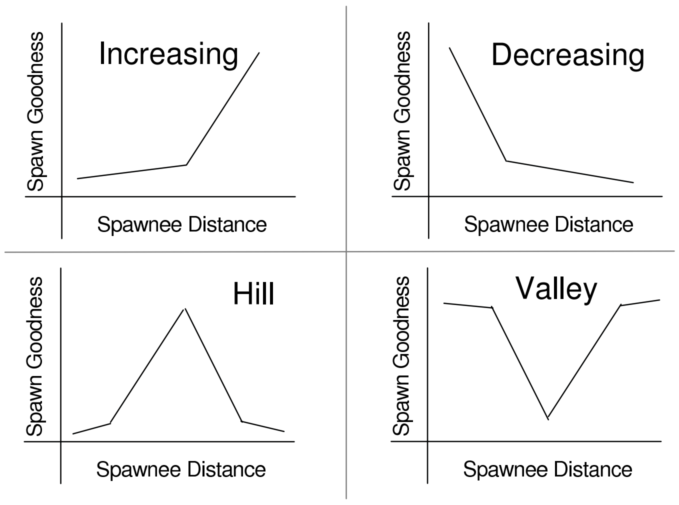


Figure 5: The *spawn goodness* profiles. We identify four major spawnee profiles given a spawner. Other profiles can be realized by composing these four. Data-dependences are a deciding factor in spawn goodness. Spawns that delay critical dataflow by introducing inter-task dependences are bad.

Figure 4(b) shows the outcome of spawning C from A. This is a profitable spawn that succeeds in reducing criticality. The instruction C was originally fetch-critical because of the mispredicted branch A. Spawning removed the execution of mispredicted branch A as well as fetch of instructions between A and C from the critical path.

On the other hand, spawning D from A, shown in Figure 4(c), is an unprofitable spawn that violates Rule 2 for spawn success. This is because spawning adds a large latency to the EE edge from C to D, delaying dataflow into execution of the branch mispredict D. Thus we delayed an existing near-critical EE edge due to a spawn. Note that, to simplify the diagrams, we have not shown several dataflow edges that contain a lot of slack because they don't affect the profitability of these spawns.

3.3 Dataflow Behavior and Spawn Success

Section 3.2 illustrated how the data-dependence edges in twolf impacted spawn selection. In general, we find that while there are other important factors involved, dataflow plays a major role in deciding the best spawn choice. The best spawn choice is not necessarily the first or the last (when ranked in distance from spawner) option available for a particular spawner. In fact, we find that the candidate spawns frequently follow a fixed number of patterns in terms of their profitability.

We define the measure of the amount by which a task spawn speeds up the program critical path as the *goodness* of that spawn choice. Based on the impact of data-dependences, we have observed several goodness profiles for spawnee choices at a given spawner. Figure 5 summarizes the basic template profiles. Most of the observed goodness profiles can be summarized through either these templates or a composition. Next we explain program behavior that causes a particular profile.

A monotonically increasing profile means that the data-dependences flowing into nearby regions have less slack than edges flowing to nodes that are more distant. A common example is code where there is limited parallelism in inner loop regions due to long serial dependence chains, but parallelism exists in the outer loop at a coarser granularity. Previous work such as [18] has used an outer-loop iteration spawn policy with some success in such cases. On the other hand, monotonically decreasing goodness occurs in cases of fine-grained parallelism. Critical values are produced in nearby regions in such cases, and spawning over them makes this dataflow inter-task, thus degrading performance.

An example of a valley goodness profile was shown in Figure 3. In that example, spawning the closest CI point (B) and spawning a distant point (G) were both profitable. In a hill profile, on the other hand, neither a closest nor a distant spawn choice is a good one. Instead the best choice lies in the intermediate region. In the next section, we describe a spawn selection policy that makes a good spawn selection for all of the above cases.

4 A Criticality-Aware Spawn Policy

Section 3 described the reasoning behind the rules for spawn success. In this section, we describe one possible implementation of a spawn selection policy that leverages those insights. The policy uses an extension of the token-passing mechanism described in Section 2.1. The next section briefly describes how we characterize spawn behavior using the token-passing mechanism. Section 4.2 describes how we make good spawn selection decisions. In Section 5, we use this policy to confirm that criticality indeed makes an impact on spawn goodness.

4.1 Spawn Assessment Mechanism

The two rules for spawn success pose the following requirements for a spawn policy:

1. Decide whether a potential spawnee is fetch-critical.
2. Decide if dataflow edges crossing the task boundary (due to a potential spawnee) don't have enough slack for the spawn to be profitable.

For the first requirement, we can monitor fetch-criticality using the unmodified token-passing predictor. A simpler heuristic that we have found to work equally well is that postdominators of low-confidence branches are very likely to be fetch-critical, since these branches are likely to mispredict (and thus be FCGEs). A similar heuristic might be used for other FCGEs.

In order to meet the second requirement, we use the following observation:

- If a delayed inter-task EE edge (due to a spawn) is part of a chain of last-arriving edges that flows into a previously critical (or near-critical) execute node, then doing the spawn most likely made the critical path worse.

If an execute node was previously critical or near-critical, it had very little slack, and its execution can not be delayed by much without affecting program performance. Therefore, if the last-arriving path into the node's execution included a delayed inter-task data-dependence edge, the execution of that node was likely delayed by more than its (tiny) slack, making the critical path worse. Such a spawn can probably be ruled out.

The original token passing mechanism can be used to detect if an instruction is execute-critical. This is done by planting a token at the execute node of an instruction, and observing if it propagates for long. By randomly sampling different instructions, a criticality profile can be constructed for relevant instructions after some training.

To detect violations of Rule 2, we modify the token-passing mechanism as follows. We randomly plant tokens in one or more of the instructions in the region between the spawner and spawnee points. Token-propagation proceeds as in the original mechanism. If one such planted token makes its way into a critical execution node in the spawnee (as identified earlier), this spawn decision probably made the critical path worse.

So, for example, when the system spawns D from spawner A as shown in Figure 4(c), a token could be planted in instructions B or C. If a token is planted in C's execution node, it would flow into D's execution node (since C $\rightarrow$ D EE edge is a last arriving edge). Since D's execution was previously identified to be on the critical path, the mechanism (correctly) claims a violation of Rule 2 in this case.

4.2 Selection Policy

We use the mechanism of Section 4.1 to approximate the goodness of a spawn choice. Spawns that violate Rule 2 with high frequency are unlikely to be profitable. When Rule 2 is violated we also record how far from the start of the spawnee thread the violation occurred. Spawn choices where violations occur close to the spawnee point are likely to be worse than those where these violations are further out. This is because the further a token must travel on a critical path to find an execute-critical node, the less likely it is that the current path was the original critical path and the more likely that current path is shorter than the original path.

Other factors such as spawn lengths are also important. Long spawns are attractive because they can jump over a large number of FCGEs at once, potentially removing all of them from the program critical path. They also help amortize the start-up and communication overheads over a long task, and are less likely to suffer load-imbalances. In particular, our architecture imposes a penalty for spawn initiation. To avoid the case where extremely small spawns degrade performance due to this penalty, we reject spawns that are too short. In addition, our machine must buffer all speculative state. Thus, we profile off line to prune out spawn pairs where the average length exceeds the modelled reorder buffer size.

We have a training phase, when we try to learn the behavior of different spawn choices. For a given spawner, we randomly select an available spawnee choice, and plant tokens to learn the the frequency of violation of Rule 2, as well as the average distance to the violation. At the end of the training phase, we approximate the goodness of a spawn choice using the metric

$$\frac{\text{average distance to violating node}}{\text{fraction of planted tokens that caused violations}}.$$

In addition to the above metric, we use a minimum threshold to turn off spawning if no profitable choice exists. For each spawner, the spawnee choice with the highest goodness is selected. The spawn decisions made at the end of the training phase are used in the evaluation phase.

As mentioned above, our objective is to confirm the importance of criticality to spawn selection, rather than to come up with the best possible spawn policy from either a performance or a hardware-implementability point of view. Therefore we idealize the training requirements on execute-criticality as well as spawn goodness. Coming up with better goodness metrics to improve performance, as well as realizable fully dynamic schemes is a subject of future work.

5 Experimental Results

In this section, we evaluate the performance of a criticality-aware spawn policy on our control-independence architecture and try to understand the results. We first describe the machine architecture that we model in Section 5.1 and our simulation infrastructure in Section 5.2. Next, we evaluate and analyze the performance of a set of spawn policies in Section 5.3.

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 2: Pipeline parameters.

5.1 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 2. Each core has local L1 instruction and data caches and branch predictors. The L1 data caches are kept coherent using an update-based protocol: a retired store is broadcast to all caches. Note that in Speculative Multithreaded processors, at any point in time, only one core is retiring instructions and committing stores to memory.

When a core fetches an instruction which is specified as a spawner, it sends a spawn command to its neighbor with the program counter of the spawnee. Inter-core register communication happens on a register-value bus. In each core, dependent instructions which are waiting for register or memory values from a predecessor thread to arrive are stored in a FIFO called the *Divert Queue*, similar to the structure used by Al-Zawawi et al [3]. Data-independent instructions are dispatched to the scheduler.

We use a dynamic mechanism similar to Skipper[5] to dynamically identify and handle inter-thread register dependences. Treating store sets[6] as architectural registers, we use the same mechanism for inter-thread memory depen-

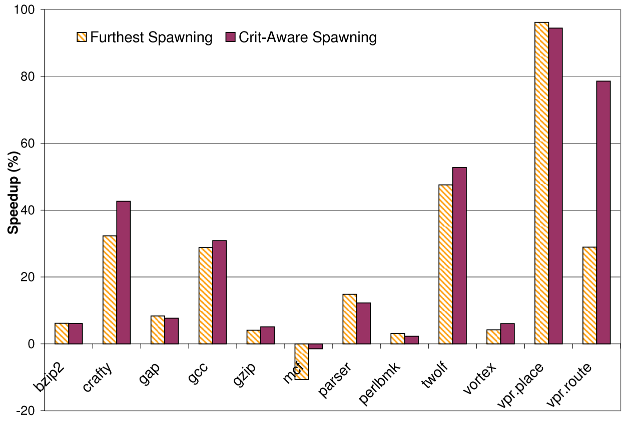


Figure 6: Percent speedup over superscalar for furthest spawn policy versus criticality-aware spawn policy. Furthest spawning misses cases of monotonically decreasing and hill shaped goodness profiles.

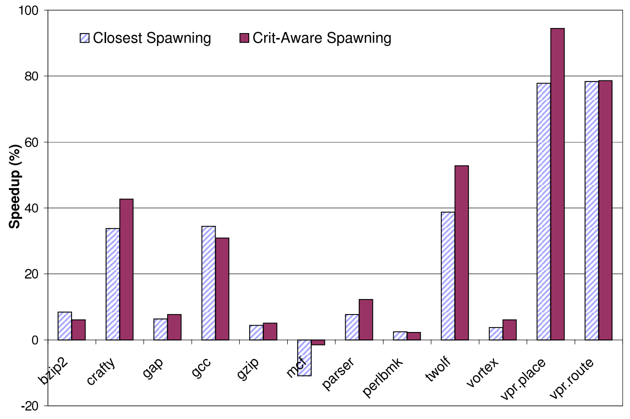


Figure 7: Percent speedup over superscalar for closest spawn policy versus criticality-aware spawn policy. Closest spawning misses out where goodness profile is monotonically increasing or hill-shaped.

dences. Each core still has a local store queue. All stores write their value 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 producer store in the spawner thread executes after a dependent load in the spawnee thread, a global load-queue detects the violation in the cycle after the store completes execution.

Each core has a large, aggressive, tournament branch predictor (192KB). The branch predictors of all cores are trained by retiring branches. When a thread is spawned, it starts off with the global history register being cleared. On most benchmarks, clearing the global history can cause a drop in branch predictor accuracy. We observe an increase in the average branch mispredict rate from 6.06% to 6.34% on the 12 simulated benchmarks.

5.2 Simulation Methodology

Our experimental evaluation was performed on an 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.

Spawn points that we use are obtained from a control-independence analysis performed on the program binary, followed by profiling to identify postdominators of low-confidence branches. A spawn cache could be used to store these postdominators[1]. However, since the total number of distinct postdominators are less than 300 for all applications, we don't model capacity or size constraints for the spawn cache.

We present results from running the SPEC2000 integer benchmarks. Our tool chain is incapable of compiling eon. All our experiments are run on the lgred input sets [16]. The simulator fast forwards through the initialization phase of all benchmarks, and executes 50 million instructions. The performance graphs that we present show the speedup of different Speculative Multithreaded configurations over a superscalar with the same configuration as a single core of the Speculative Multithreaded processor.

5.3 Performance of Spawn Policies

Next we evaluate a set of spawn policies on the above described architecture. For all three policies analyzed here, we start out with the elimination strategies that prune out spawns that are unlikely to be profitable. This includes shortlisting postdominators of low-confidence branches, and profiling spawn lengths to impose a minimum and maximum average distance threshold, as described in Section 4.2. This is a very aggressive pruning strategy, that straightaway eliminates a large chunk of likely unprofitable spawns.

Having shortlisted the likely successful spawn choices, we next evaluate the performance of three different spawn policies. Two policies are natural contenders: spawn the closest available choice, and spawn the furthest available choice. The closest spawn choice represents a highly tuned (due to the pruning steps) version of the skipper spawn policy[5]. The furthest policy is attractive for overhead amortization and load-balancing reasons. These policies are attractive if successful, since these don't require involved analysis of inter-task data dependences. The third policy evaluated is the criticality-aware spawn policy described in Section 4.2.

Figures 6 and 7 show the performance obtained from the first two policies compared to the data criticality-aware policy. The first observation is that the closest and furthest point spawn policies lead to a high base speedup over the superscalar processor. This is because of our aggressive offline pruning techniques that are tuned for our evaluation architecture. Further, each of these policies make a good se-

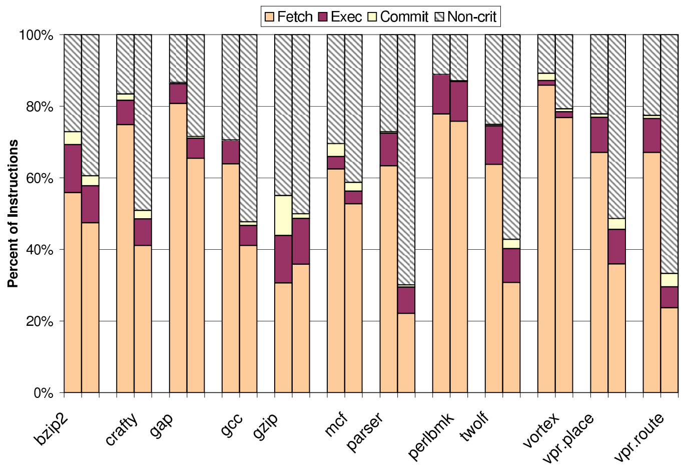


Figure 8: Breakup of instruction criticality on a superscalar (left bar), compared to criticality-aware spawning (right bar). Spawning significantly decreases fetch criticality.

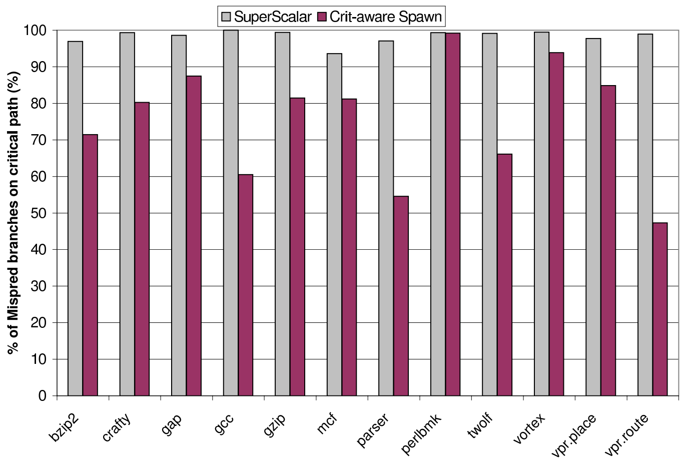


Figure 9: Percent of mispredicted branches that are on the program critical path for superscalar and spawning. Spawning removes a large number of branch mispredicts from the program critical path.

lection for at least two of the four spawn goodness profiles described in Section 3.3. In particular, the closest spawn policy will make a good choice for monotonically decreasing and valley profiles, while furthest spawning is likely to work for monotonically increasing and valley profiles.

The performance numbers reflect this intuition. For benchmarks such as VPR Place where parallelism exists at coarse granularities (rather than at the innermost loop level), furthest spawning performs better than closest spawning. On the other hand, furthest spawning is worse where decreasing profile dominates and fine-grained parallelism exists, as shown in results for benchmarks like VPR Route.

Figures 6 and 7 also show that while closest and furthest spawning capture some of the cases, they are not robust heuristics. Criticality-aware spawn policy, on the other hand, tries to capture the goodness profile on a per-spawn basis, and approaches the best heuristic for most benchmarks. Further, there are benchmarks such as twolf and crafty where this policy outperforms even the better of the previous two heuristics. These are cases that have mixed parallelism profiles across their spawn points, including hill profiles that are not captured by either of the above two. Also note that this policy doesn't target FCGEs like resource stalls and instruction misses, hence it doesn't get good performance on benchmarks like bzip2, gap, gzip, mcf and vortex, where branch mispredicts are not the dominant fetch-criticality generating effect.

Finally, we show the impact of spawning on fetch-criticality of instructions in Figure 8. The figure shows the breakup of instruction-criticality for superscalar and CI scenarios, calculated using the trace-based analysis as described in Section 2.1. The trace was obtained using a timing model [39] that closely approximates the architecture evaluated. Instructions not on the critical path are classified as non-critical. We find that spawning significantly reduces fetch-criticality without significantly degrading execute-criticality in these applications by fetching instructions that are control independent of FCGEs earlier than before.

Spawning over FCGEs also helps remove some of them from the program critical path. This is shown in Figure 9 for mispredicted branches. We find that while most of the mispredicted branches lie on the program critical path in superscalar execution, our spawn choices made as many as 50% (for VPR Route) of those non-critical. Also note that the reduction in fetch-criticality doesn't necessarily track performance improvement, since that depends on how close the next near-critical path is.

6 Related Work

This paper develops a simple model of microarchitectural criticality for control-independence architectures, that removes artificial fetch dependences enforced by superscalar processors. This model is based directly on the model from Fields, Rubin and Bodík [9], but also influenced by a variety of other models [4, 15, 34, 37]. In the superscalar domain criticality information has been used to drive resource allocation decisions. Critical path information can be used to reduce the power consumption of instructions that are not on the critical path [25, 31], to direct non-critical instructions to slower functional units [8], and to drive steering decisions in clustered machines [30].

The paper then uses the insight generated by the criticality model to design a spawn selection policy for control-independence architectures. There has been previous work in task selection. The Multiscalar compiler performs task selection by walking the static program control-flow graph (CFG) and partitioning using heuristics that incorporate task size, intertask control flow, and intertask data dependences [33, 38]. A later work [14] annotates the static CFG with edge weights that depend on task size, and partitions using the min-cut algorithm to minimize the value communication between tasks. The Mitosis compiler [26] builds upon the idea of Control-Quasi Independence [20] to select tasks, while ensuring a minimum task length. However, none of these approaches take into account the criticality of dependences that cross task boundaries when making task selection, while we find that doing so can improve performance.

Previous work by our group [19] demonstrated that applications exhibit substantial amounts of branch-mispredict level parallelism (BLP), and that speculative multithreading architectures can overlap the execution of multiple independent mispredicted branches. The criticality approach presented in this paper generalizes heuristics like BLP or MLP. More importantly, the token-passing based approach described in this paper could be used to realize a fully dynamic system.

Several other groups have used criticality information to drive policies in speculatively multithreaded processors. Nagpal and Bhowmik add latency to non-critical load instructions that might otherwise cause inter-thread data misspeculation [23]. Tuck et al used a task level, rather than instruction level, dynamic criticality analysis to drive task scheduling for speculative multithreading [36]. Our work attacks thread partitioning rather than instruction or task scheduling and so required a more accurate criticality model.

In this paper we have focussed on exploiting control independence with speculatively multithreaded or thread-level speculative processors [1, 2, 12, 17, 18, 21, 24, 29, 33, 35]. The closest architecture to ours, in terms of conservatively adding delay to inter-task data dependences is the Skipper architecture [5]. However, Skipper limited itself to spawning the closest postdominator, while we show that this spawn choice might be suboptimal. Skipper also limits itself to a single fetch unit. We believe our criticality model can be used to draw insights about architectures such as Skipper that are similar but not identical to ours.

Finally, squash reuse techniques also leverage control independence [3, 10, 13, 28, 32]. However, these processors present a different set of tradeoffs than do the speculatively multithreaded or thread-level speculative processors. In particular, they only have a single fetch unit, and further they are only able to exploit control-independence points that the fetch unit actually reaches.

7 Conclusions and Future Directions

Superscalar processors impose the restrictions of in-order fetch, which becomes a major bottleneck to performance. Control-independence architectures remove this restriction, and hence promise to improve performance by reducing the impact of *fetch criticality generating events* (FCGEs). However, successful parallelization requires striking the right trade-offs between fetch-criticality and execute-criticality.

In this paper, we present a model to reason about control-independence parallelization, and describe how it can be used to analyze the trade-off described above. We find that spawn selection is one of the deciding factors in successful parallelization, and that there are two simple rules that give conditions for a successful task spawn. These rules naturally lend themselves to a spawn selection policy, and we describe the policy as well as a mechanism that could be used to realize the policy in a real architecture. The policy is designed to select spawns that don't degrade the critical path due to delayed inter-task data dependences.

In addition, we find that the observed spawnee goodness behavior can be summarized through four common profiles. Evaluations show that simple policies that exploit only a subset of these profiles miss out on potential performance, and that a criticality-aware policy is robust across different dataflow behavior. Finally, detailed analysis of the performance numbers confirms that the spawn policy described above, indeed reduces fetch-criticality in applications.

Reasoning in this framework also opens up exciting future directions. This work currently deals with one category of FCGEs, namely mispredicted branches. A straightforward extension of this work would be to the other 2 classes of FCGEs, which improve performance on other classes of applications. Further, for some of the applications analyzed in this paper, alleviating fetch-criticality doesn't improve performance much, because it exposes other previously near-critical paths along long-spanning EE edges as critical, or exposes limitations of the back end architecture. This opens up the possibility of focusing program transformations on the new critical paths to remove EE bottlenecks, and for exploring better back end architectures.

Acknowledgments

We thank Steve Lumetta, Naveen Neelakantam, Sanjay Patel, Yale Patt, Pradeep Ramachandran, Pierre Salverda, and the anonymous reviewers for valuable feedback. The work reported in this paper was supported in part by the National Science Foundation under grant CCR-0429711 and CCF-0746910 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 Univ. of Illinois.

Bibliography

[1] M. Agarwal, K. Malik, K. M. Woley, S. S. Stone, and M. I. Frank. Exploiting postdominance for speculative parallelization. *High Perf 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), 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. *Intl Symp Comp Arch*, (ISCA-25):142–153, 1998.

[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] B. Fields, R. Bodík, and M. Hill. Slack: Maximizing performance under technological constraints. *Int'l Symp Comp Arch*, (ISCA-29), 2002.

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

[10] A. Gandhi, H. Akkary, and S. T. Srinivasan. Reducing branch misprediction penalty via selective branch recovery. *Intl Symp High Performance Comp Arch*, (HPCA-10):254, 2004.

[11] A. Garg, M. W. Rashid, and M. Huang. Slackened memory dependence enforcement: Combining opportunistic forwarding with decoupled verification. *Intl 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), 2007.

[14] T. A. Johnson, R. Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Prog. Lang. Design and Implementation*, pages 59–70, 2004.

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

[17] V. Krishnan and J. Torrellas. A chip multiprocessor architecture with speculative multithreading. *IEEE Trans. Comput.*, 47(9), September 1999.

[18] 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.

[19] K. Malik, M. Agarwal, S. S. Stone, K. M. Woley, and M. I. Frank. Branch-mispredict level parallelism for control independence. *High Perf Comp Arch*, (HPCA-14), 2008.

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

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

[22] A. I. Moshovos. *Memory dependence prediction*. PhD thesis, University of Wisconsin-Madison Computer Sciences Department, 1998.

[23] R. Nagpal and A. Bhowmik. Criticality driven energy aware speculation for speculatively multithreaded processors. In *International Conference of High-Performance Computing*, volume 12, pages 19–28, Dec 2005.

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

[25] R. Pyreddy and G. Tyson. Evaluating design tradeoffs in dual speed pipelines. In *Workshop on Complexity Effective Design*, 2001.

[26] C. G. Quinones, C. Madriles, J. Sanchez, P. Marcuello, A. Gonzalez, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. *Prog. Lang. Design and Impl*, pages 269–279, 2005.

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

[28] 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.

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

[30] P. Salverda and C. Zilles. A criticality analysis of clustering in superscalar processors. *Intl Symp Microarchitecture*, (MICRO-38):55–66, 2005.

[31] J. Seng, E. Tune, and D. Tullsen. Reducing power with dynamic critical path information. *Intl Symp Microarchitecture*, (MICRO-34):114–123, 2001.

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

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

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

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

[36] J. Tuck, W. Liu, and J. Torrellas. CAP: Criticality analysis for power-efficient speculative multithreading. *Intl Conf Computer Design*, (ICCD), 2007.

[37] E. Tune, D. Liang, D. Tullsen, and B. Cler. Dynamic prediction of critical path instructions. *Intl Symp High-Perf Comp Arch*, (HPCA-7):185–195, 2001.

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

[39] D. W. Wall. Limits of instruction-level parallelism. Research Report 93/6, DEC Western Research Laboratory, Nov. 1993.