Focusing Processor Policies via Critical-Path Prediction

Brian Fields Shai Rubin Rastislav Bodík

Computer Sciences Department
University of Wisconsin–Madison
{fields,shai,bodik}@cs.wisc.edu

Abstract

Although some instructions hurt performance more than others, current processors typically apply scheduling and speculation as if each instruction was equally costly. Instruction cost can be naturally expressed through the *critical path*: if we could predict it at run-time, egalitarian policies could be replaced with cost-sensitive strategies that will grow increasingly effective as processors become more parallel.

This paper introduces a hardware predictor of instruction criticality and uses it to improve performance. The predictor is both effective and simple in its hardware implementation. The effectiveness at improving performance stems from using a dependence-graph model of the microarchitectural critical path that identifies execution bottlenecks by incorporating both data and machine-specific dependences. The simplicity stems from a token-passing algorithm that computes the critical path without actually building the dependence graph.

By focusing processor policies on critical instructions, our predictor enables a large class of optimizations. It can (i) give priority to critical instructions for scarce resources (functional units, ports, predictor entries); and (ii) suppress speculation on non-critical instructions, thus reducing "useless" misspeculations. We present two case studies that illustrate the potential of the two types of optimization, we show that (i) critical-path-based dynamic instruction scheduling and steering in a clustered architecture improves performance by as much as 21% (10% on average); and (ii) focusing value prediction only on critical instructions improves performance by as much as 5%, due to removing nearly half of the misspeculations.

1 Introduction

Motivation. Even though some instructions are more harmful to performance than others, current processors employ egalitarian policies: typically, each load instruction, each cache miss, and each branch misprediction are treated as if they cost an equal number of cycles. The lack of focus on bottleneck-causing (*i.e.*, critical) instructions is due to the difficulty of identifying the effective cost of an instruction. In particular, the local view of the execution that is inherent in the processor limits its ability to determine the effects of instruction overlap. For example: "Does a 'bad' long-latency instruction actually harm the execution, or is it made harmless by a chain of 'good' instructions that completely overlap with it?"

A standard way to answer such questions in a parallel system is *critical-path analysis*. By discovering the chain of dependent events that determined the overall execution time, critical-path analysis has been used successfully for identifying performance bottlenecks in large-scale parallel systems, such as communication networks [3, 9].

Out-of-order superscalar processors are fine-grain parallel systems: their instructions are fetched, re-ordered, executed, and committed in parallel. We argue that the level of their parallelism and sophistication has grown enough to justify the use of critical-path analysis of their microarchitectural execution. This view is shared by Srinivasan and Lebeck [18], who computed an indirect measure of the critical path, called *latency tolerance*, that provided non-trivial insights into the parallelism in the memory system, such as that up to 37% of L1 cache hits have enough latency tolerance to be satisfied by a lower-level cache.

The goal of this paper is to exploit the critical path by making processor policies sensitive to the actual cost of microarchitectural events. As was identified by Tune *et al.* [21], a single critical-path predictor enables a broad range of optimizations in a modern processor. In this paper, we develop optimizations that fall into two categories:

- *Resource arbitration:* Resources can be better utilized by assigning higher priority to critical instructions. For example, critical instructions can be scheduled before non-critical ones whenever there is contention for functional units or memory.
- *Misspeculation reduction.* The risk of misspeculations can be reduced by restricting speculation to critical instructions. For instance, value prediction could be applied only to critical instructions. Because, by definition, it is pointless to speed up non-critical instructions, speculating them brings risk but no benefit.

The Problem. The analytical power of the critical path is commonly applied in compilers for improving instruction scheduling [15, 16], but has been used in the microarchitectural community only as an informal way of describing inherent program bottlenecks. There are two reasons why the critical path is difficult to exploit in microprocessors. The first is the **global** nature of the critical path: While compilers can find the critical path through examination of the dependence graph of the program, processors see only a small window of instructions at any one time.

The second reason is that the compiler's view of the critical path, consisting merely of data dependences, does not

1063-6897/01 $10.00 © 2001 IEEE

precisely represent the critical path of a program executing on a particular processor implementation. A real processor imposes **resource constraints** that introduce dependences beyond those seen by a compiler. A finite re-order buffer, branch mispredictions, finite fetch and commit bandwidth are all examples of resources that affect the critical path.

One method for resolving these two problems is to identify the critical path using *local*, but *resource-sensitive* heuristics such as marking the oldest uncommitted instructions [21]. Our experiments show that for some critical-path-based optimizations, these heuristics are an inaccurate indicator of an instruction's criticality. Instead, optimizations seem to require a global and robust approach to critical-path prediction.

Our Solution. In order to develop a robust and efficient hardware critical-path predictor, we divided the design tasks into two problems: (**1**) development of a *dependence-graph model* of the critical path; and (**2**) a predictor that follows this model when learning the critical path at run-time.

Our *dependence-graph model* of the critical path is simple, yet able to capture in a uniform way the critical path through a given *microarchitectural* execution of the program. The model represents each dynamic instruction with three nodes, each corresponding to an event in the lifetime of the instruction: the instruction being *dispatched* into the instruction window, *executed*, and *committed*. Edges (weighted by latencies) represent various data and resource dependences between these events during the actual execution. Data dependences connect *execute* nodes, as in the compiler's view of the critical path. A resource dependence due to a mispredicted branch induces an edge from the *execute* node of the branch to the *dispatch* node of the correct target of the branch; other resource dependences are modeled similarly. Our validation of the model indicates that it closely reflects the critical path in the actual microexecution.

Although we developed the model primarily to drive our predictor, it can be used for interesting performance analyses. For example, thanks to its 3-event structure, the critical path determines not only whether an instruction is critical, but also why it is critical (*i.e.*, is fetching, executing, or committing of the dynamic instruction its bottleneck?).

The hardware *critical-path predictor* performs a global analysis of the dependence graph. Given the dependence-graph model, we can compute the critical path simply as the longest weighted path. A simple graph-theory trick allows us to examine the graph more efficiently, *without* actually building it: When training, our predictor plants a token into a dynamic instruction and propagates it forward through certain dependences; if the token propagates far enough, the seed node is considered to be critical. The predictor is also adaptable: an instruction can be re-trained by re-planting the token. We show how this algorithm can be implemented with a small array and simple control logic.

To study the usefulness of our predictor, we selected two optimizations (one from each of the above categories): cluster scheduling and value prediction. We found the predictor not only accurately finds the critical path, but also *consistently* improves performance.

In summary, this paper presents the following contributions:

- A validated model that exposes the critical path in a microarchitectural execution on an out-of-order processor. The model treats resource and data dependences uniformly, enhancing and simplifying performance understanding.
- An efficient token-based predictor of the critical path. Our validation shows that the predictor is very precise: it predicts criticality correctly for 88% of all dynamic instructions, on average.
- We use our criticality predictor to *focus the scheduling policies* of a clustered processor on the critical instructions. Our predictor improves the performance by as much as 21% (10% on average), delivering nearly an order of magnitude more improvement than critical-path predictors based on *local* heuristics.
- As a proof of concept that the critical-path predictor can optimize speculation, we experimented with *focused value prediction*. Despite the low misprediction rate of our baseline value predictor, focusing delivered speedups of up to 5%, due to nearly halving the amount of value mispredictions.

The next section describes and validates our model of the critical path. Section 3 presents the design, implementation, and evaluation of the predictor built upon the model. Section 4 uses the predictor to focus instruction scheduling in cluster architectures and value prediction. Finally, Section 5 relates this paper to existing work and Section 6 outlines future directions.

2 The Model of the Critical Path

This section defines a dynamic dependence graph that serves as a model of the microexecution. We will use the model to profile (in a simulator) the critical path through a trace of dynamic instructions. In Section 3, we will use the model to predict the critical path, *without* actually building a dependence graph.

In compilers, the run-time ordering of instructions is modeled using a program's inherent *data* dependences: each instruction is abstracted as a single node; the flow of values is represented with directed edges. Such a *machine-independent* modeling misses important *machine-specific* resource dependences. For example, a finite re-order buffer can fill up, stalling the fetch unit. As we show later in this section, such dependences can turn critical data dependences into critical resource dependences.

We present a model with sufficient detail to perform critical-path-based optimizations in a typical out-of-order processor (see Table 5). Our critical-path model accounts for the effects of branch mispredictions, in-order fetch, in-order commit, and a finite re-order buffer. If a processor implementation imposes significantly different constraints, such as out-of-order fetch [19, 14], new dependences can be added to our model, after which our techniques for computing and predicting the critical path can be applied without change.

Our model abstracts the microexecution using a dynamic dependence graph. Each dynamic instruction i is represented by three nodes: the dispatch node D_i, the execute node E_i, and the commit node C_i. These three nodes denote events within the machine pertaining to the instruction: the instruction being dispatched into the instruction window, the instruction becoming ready to be executed, and the instruction committing. Directed edges connect dependent nodes. We explicitly model

name	constraint modeled	edge	
DD	In-order dispatch	$D_{i-1} \rightarrow D_i$	
CD	Finite re-order buffer	$C_{i-w} \rightarrow D_i$	w = size of the re-order buffer
ED	Control dependence	$E_{i-1} \rightarrow D_i$	inserted if $i-1$ is a mispredicted branch
DE	Execution follows dispatch	$D_i \rightarrow E_i$	
EE	Data dependences	$E_j \rightarrow E_i$	inserted if instruction j produces an operand of i
EC	Commit follows execution	$E_i \rightarrow C_i$	
CC	In-order commit	$C_{i-1} \rightarrow C_i$	

Table 1: **Dependences captured by the critical-path model**, grouped by the target of the dependence.

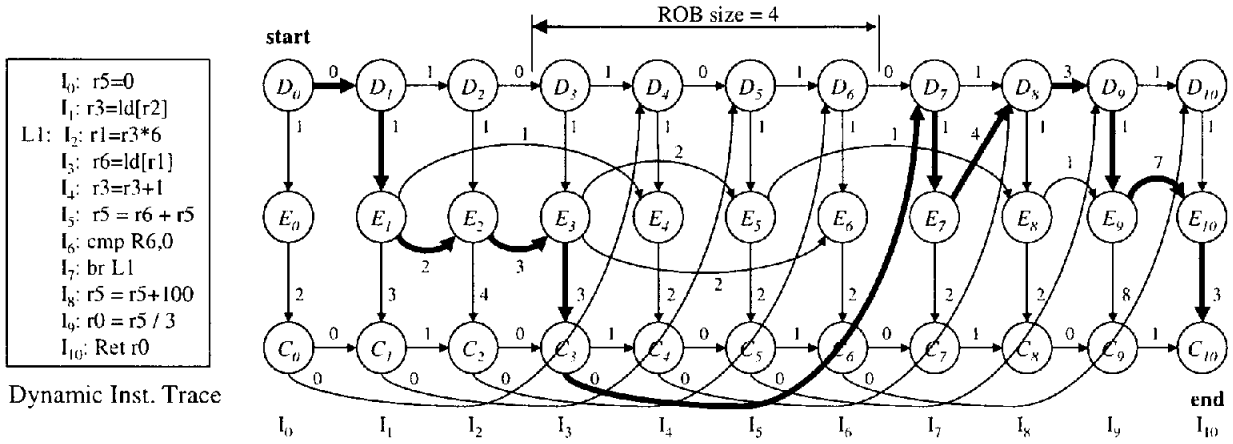


Figure 1: **An instance of the critical-path model from Table 1.** The dependence graph represents a sequence of dynamic instructions. Nodes are events in the lifetime of an instruction (the instruction being dispatched, executed, or committed); the edges are dependences between the occurrences of two events. A weight on an edge is the latency to resolve the corresponding dependence. The critical path is highlighted in bold.

seven dependences, listed in Table 1 and illustrated in Figure 1. We will describe each edge type in turn.

Data dependence, *EE*, edges are inserted between *E* nodes. An *EE* edge from instruction j to instruction i introduces a constraint that instruction i may not execute until the value produced by j is ready. Both register dependences and dependences through memory (between stores and loads) are modeled by these edges. The *EE* edges are the only dependences typically modeled by a compiler.

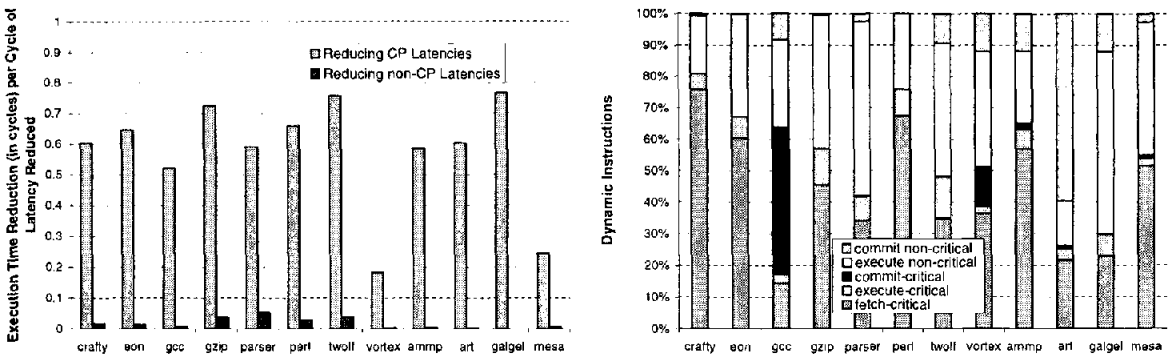
Modeling the critical path with microarchitectural precision is enabled by adding *D*-nodes (instruction being dispatched) and *C*-nodes (instruction being committed). The intra-instruction dependences *DE* and *EC* enforce the constraint that an instruction cannot be executed before it is dispatched, and that it cannot be committed before it finishes its execution. In our out-of-order processor model, instructions are dispatched in-order. Thus, a dependence exists between every instruction's *D* node and the immediately following—in program order—instruction's *D* node. This dependence is represented by *DD* edges. Similarly, the in-order commit constraint is modeled with *CC* edges.

So far we have discussed the constraints of data dependences, in-order dispatch, and in-order commit. Now we will describe how we model two other significant constraints in out-of-order processors: branch mispredictions and the finite re-order buffer. A branch misprediction introduces a constraint that the correct branch-target instruction cannot be dispatched until after the mispredicted branch is resolved (*i.e.*, executed). This constraint is represented by an *ED* edge from the *E* node of the mispredicted branch to the *D* node of the first instruction of the correct control-flow path. An example of a mispredicted-branch edge can be seen between instructions I_7 and I_8 of Figure 1. Note that it is not appropriate to insert *ED* edges for correctly predicted branches because a correct prediction effectively breaks the *ED* constraint, by permitting the correct-path instructions to be fetched and dispatched (*D*-node) before the branch is resolved (*E*-node). Also note that we do not explicitly model wrong-path instructions. We believe these instructions have only secondary effects on the critical path (*e.g.*, they could cause data cache prefetching). Our validation, below, shows that our model provides sufficient detail without modeling such effects.

The re-order buffer (ROB) is a FIFO queue that holds instructions from the time they are dispatched until they have committed. When the ROB fills up, it prevents new instructions from being dispatched into the ROB. To impose the constraint that the oldest instruction in the ROB must be committed before another instruction can be dispatched, we use *CD* edges. In a machine with a four-instruction ROB, *CD* edges span four dynamic instructions (see Figure 1).

The edge weights reflect the actual microexecution. Each weight equals the time it took to resolve the particular dynamic instance of the dependence. For instance, the weight of an *EE* edge equals the execution latency plus the wait time for the functional unit. Thus, the weight may be a combination of multiple sources of latency. Note that while these dynamic latencies are part of the model, we do not need to measure their actual values. Instead, one of the contributions of this paper is to show how to compute the critical path by merely *observing* the order in which the dependences are resolved (see Section 3.1).

Given a weighted graph of all the dynamic instructions in a



(a) Validation of the critical-path model.

(b) Breakdown of the dynamic instruction count.

Figure 2: **Validation of the model and instruction count breakdown.** **(a)** Comparison of the performance improvement from reducing critical latencies vs. non-critical latencies. The performance improvement from reducing critical latencies is much higher than from non-critical latencies, demonstrating the ability of our model to differentiate critical and non-critical instructions. **(b)** The breakdown of instruction criticality. Only 26–80% of instructions are critical for any reason (fetch, execute, or commit) and only 2–13% are critical because they are executed too slowly.

program, the critical path is simply the longest *weighted path* from the D node of the first instruction to the C node of the last instruction. The critical path is highlighted in bold in Figure 1.

Let us note an important property of the dependence-graph model: No critical-path edge can span more instructions than the ROB size (ROB_size). The only edges that could, by their definition, are EE edges. An EE edge of such a length implies the producer and the consumer instructions are not in the ROB at the same time. Thus, by the time the consumer instruction is dispatched into the ROB, the value from the producer would be available, rendering the dependence non-critical. This is an important observation that we will exploit to bound the storage required for the predictor's training array without any loss of precision.

Validating the Model. Next, we validate that our model successfully identifies critical instructions. This validation is needed because the hardware predictor design described in the next section is based on the model, and we want to ensure the predictor is built upon a solid foundation.

Our approach to validation measures the effect of *decreasing*, separately, the execution latencies of critical and non-critical instructions. If the model is accurate, decreasing critical instruction latencies should have a big impact on performance: we are directly reducing the length of the critical path. In contrast, decreasing noncritical instruction latencies should *not* affect performance at all since we have not changed the critical path.

Since some instructions have an execution latency of one cycle, and our simulator does not support execution latencies of zero cycles, we established a baseline by running a simulation where all the latencies were increased by one cycle (compared to what is assumed in the simulator configuration of Section 4, Table 5). The critical path from this baseline simulation was written to disk. We then ran two simulations that each read the baseline critical path and decreased all critical (non-critical) latencies by one cycle, respectively. The resulting speedups are plotted in Figure 2(a) as cycles of execution time reduction per cycle of latency decreased.

The most important point in this figure is that the performance improvement from decreasing critical path latencies is much larger than from decreasing non-critical latencies. This indicates that our model, indeed, identifies instructions critical to performance.

Note that even though we are directly reducing critical path latencies, not every cycle of latency reduction turns into a reduction of a cycle of execution time. This is because reducing critical latencies can cause a *near-critical* path to emerge as the new critical path. Thus, the magnitude of performance improvement is an indication of the degree of dominance of the critical path. From the figure, we see that the dominance of the critical path varies across the different benchmarks. To get the most leverage from optimizations, it may be desirable to optimize this new critical path as well. Our predictor, described in the next section, enables such an adaptive optimization by predicting critical as well as near-critical instructions.

Finally, there is a very small performance improvement from decreasing non-critical latencies. The reason is that our model (intentionally) does not capture all machine dependences. As a result, a dynamic instruction marked as non-critical may in fact be critical. For instance, reducing the latency of a load that was marked non-critical by the model may speed up the prefetch of a cache line needed later by a (correctly marked) critical load, which reduces the critical path. Because the cache-line-sharing dependence between the loads was not modeled, some other instruction was blamed for the criticality of the second load. Although we could include more dependences to model such constraints, the precision observed here is sufficient for the optimizations we present. It should be noted that the more complex the model, the more expensive our critical-path predictor.

Breakdown of criticality. A unique characteristic of our model is the ability to detect not only whether an instruction is critical, but also why it is critical, *i.e.*, whether fetching, executing, or committing the instruction is on the critical path. This information can be easily detected from the model. For instance, if the critical path includes the dispatch node of an instruction, the instruction is fetch-critical (the three-node model effectively collapses fetching and dispatching into the D-node). Analogous rules apply for execute and commit nodes.

To estimate potential for critical-path optimizations, a breakdown of instruction criticality is shown in Figure 2(b). We can distinguish two reasons why an instruction may be non-critical: (a) it is *execute-non-critical* if it is overlapped by

the execution latency of a critical instruction (*i.e.*, it is skipped over by a critical *EE* edge); or (b) it is *commit-non-critical* if it "sits" in the ROB during a critical ROB stall (*i.e.*, it is skipped over by a critical *CD* edge). Note that if parallel equal-length chains exist as part of the critical path, only one of the chains was included in the breakdown.

The figure reveals that many instructions are non-critical (20–74% of the dynamic instructions). This indicates a lot of opportunity for exploiting the critical path: we can focus processor policies on the 26–80% of dynamic instructions that are critical. The data is even more striking if you consider *why* the instructions are critical: only 2–13% of all instructions are critical for being *executed* too slowly. This observation has profound consequences for some optimizations. Value prediction, for instance, will not help performance unless some of this small subset of instructions are correctly predicted (a correct prediction of a critical instruction may, of course, expose some execute-noncritical instructions as critical).

3 Predicting the Critical Path in Hardware

This section presents an algorithm for efficiently computing the critical path in hardware. A naive algorithm would (1) build the dependence graph for the entire execution, (2) label the edges with observed latencies, and then (3) find the longest path through the graph. Clearly, this approach is unacceptable for an efficient hardware implementation.

Section 3.1 presents a simple observation that eliminates the need for explicit measurement of edge latencies (step 2) and Section 3.2 then shows how to use this observation to design an efficient predictor that can find the critical path without actually building the graph (steps 1 and 3).

3.1 The *Last-Arriving* Rules

Our efficient predictor is based on the observation that a critical path can be computed solely by observing the arrival order of instruction operands; no knowledge of actual dynamic latencies is necessary. The observation says that if a dependence edge $n \rightarrow m$ is on the critical path, then, in the real execution, the value produced by n must be the *last-arriving* value amongst all operands of m; if it was not the last one, then it could be delayed without any performance harm, which would contradict its criticality. (Note that if multiple operands arrive simultaneously, there are multiple *last-arriving* edges, potentially leading to parallel critical paths.) Two useful rules can be derived from this observation: (1) each edge on the critical path is a last-arriving edge; (2) if an edge is not a last-arriving edge, then it is not critical.

The last-arriving rule described above applies to the data flow (*EE*) edges. Crucial for the computation of the critical path is whether we can also define the last-arriving rules for the micro-architectural dependences.[1] It turns out that all we need is to observe simple hardware events: for example, a dispatch (*DE*) dependence is considered to arrive last if the data operands are ready when the instruction is dispatched. The remaining last-arriving rules are detailed in Table 2.

The last-arriving rules greatly simplify the construction of the critical path. The critical path can be determined by starting at the commit node of the last instruction and traversing the graph *backward* along last-arriving edges until the first instruction's dispatch node is encountered (see Figure 4). Note the efficiency of the algorithm: no edge latencies need to be tracked, and only the last-arriving subset of the graph edges must be built. This is how we *precisely* profile the critical path in a simulator. The predictor, because it computes only an *approximation* of the critical path, does not even build the last-arriving subgraph. Instead, it receives from the execution core a stream of last-arriving edges and uses them for training, without storing any of the edges (see Figure 3).

[1]The arrival order of *operands* is used only for the *E* nodes. For *D* and *C* nodes, we conveniently overload the term *last-arriving* and use it to mean the order of completion of *other* microarchitectural events.

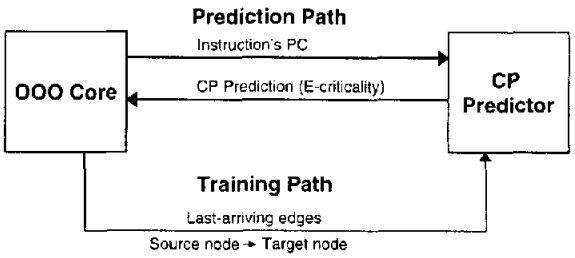


Figure 3: **The interface between the processor and the critical-path predictor.** On the training path, the core provides, for each committed instruction, the last-arriving edges into each of the instruction's three nodes (*D*, *E*, *C*). On the prediction path, the predictor answers whether a given static instruction is *E*-critical (optimizations in this paper exploit only *E*-criticality).

Before we describe the predictor, let us note that we expect that the last-arriving rules can be implemented in hardware very efficiently by "reading" control signals that already exist in the control logic (such as an indication that a branch misprediction has occurred) or observing the arrival order of data operands (information that can be easily monitored in most out-of-order processor implementations).

3.2 The Token-Passing CP Predictor

Although the algorithm described above can be used to find the critical path efficiently in a *simulator*, it is not suitable for a hardware implementation. The primary problem is that the backward traversal can be expensive to implement: any solution seems to require buffering the graph of the entire execution before we could begin the traversal.

Instead of constructing the entire graph, our predictor works on the intuitive notion that since the critical path is a chain of last-arriving edges through the entire graph, then a *long* last-arriving chain is *likely* to be part of the critical path. Thus, we predict that an instruction is critical if it belongs to a sufficiently long last-arriving chain. The important advantage of our approach is that long last-arriving chains can be found through *forward* propagation of tokens, rather than through a backwards traversal of the graph.

The heart of the predictor is a token-based "trainer." The training algorithm (see Figure 5) works through frequent *sampling* of the criticality of individual nodes of instructions. To take a criticality sample of node n, a token is planted into n (**step 1**) and propagated *forward* along *all* last-arriving edges (**step 2**). If there is more than one outgoing last-arriving edge the token is replicated. At some nodes, there may be no outgoing last-arriving edges for the token to propagate further. If all copies of the token reach such nodes, the token *dies*, indicating that node n **must not** be on the critical path, as there is definitely no chain of last-arriving edges from the beginning of the

target node	edge	last-arriving condition
D	$E_{i-1} \rightarrow D_i$	if i is the first committed instruction since a mispredicted branch.
	$C_{i-w} \rightarrow D_i$	if the re-order buffer was stalled the previous cycle.
	$D_{i-1} \rightarrow D_i$	if neither *ED* nor *CD* arrived last.
E	$D_{i-1} \rightarrow E_i$	if all the operands for instruction i are ready by the time i is dispatched.
	$E_j \rightarrow E_i$	if the value produced by instruction j is the last-arriving operand of i and the operand arrives after instruction i has been dispatched.
C	$E_i \rightarrow C_i$	if instruction i delays the in-order commit pointer (*e.g.*, the instruction is at the head of the re-order buffer but has not completed execution and, hence, cannot commit).
	$C_{i-1} \rightarrow C_i$	if edge *EC* does not arrive last (*i.e.*, instruction i was ready to commit before in-order commit pointer permitted it to commit).

Table 2: **Determining last-arriving edges.** Edges are grouped by their target node. Every node must have at least one incoming last-arriving edge. However, some nodes may not have an outgoing last-arriving edge. Such nodes are non-critical.

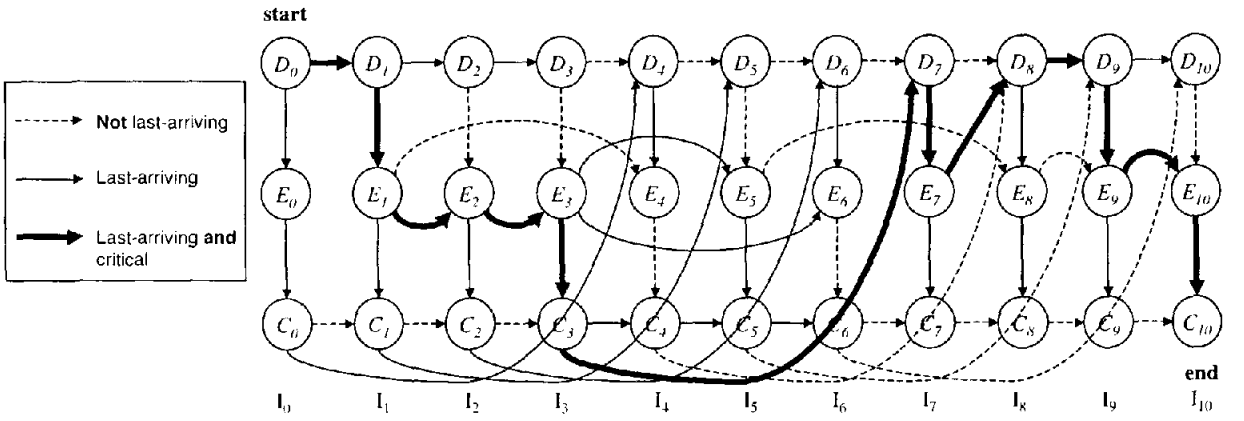


Figure 4: **The dependence graph of our running example with last-arriving edges highlighted.** The critical path is a chain of last-arriving edges from *start* to *end*. Note that some nodes have multiple last-arriving nodes due to simultaneous arrivals.

1. Plant token at node *n*.
2. Propagate token forward along *last-arriving* edges. If a node does not have an outgoing last-arriving edge, the token is not propagated (*i.e.*, it *dies*.)
3. After allowing token to propagate for some time, check if the token is still alive.
4. If token is alive, train node *n* as critical; otherwise, train *n* as non-critical.

Figure 5: **The token-passing training algorithm.**

program to the end that contains node n. On the other hand, if a token remains alive and continues to propagate, it is **increasingly likely** that node n is on the critical path. After the processor has committed some threshold number of instructions (called the *token-propagation-distance*), we check if the token is still alive (**step 3**). If it is, we assume that node n was critical; otherwise, we know that node n was non-critical. The result of the token propagation is used to train the predictor (**step 4**). Clearly, the larger the token-propagation-distance, the more likely the sample will be accurate.

Implementation. The hardware implementation consists of two parts: the critical-path table and the trainer. The critical-path table is a conventional array indexed by the PC of the instruction. The predictions are retrieved from the table early in the pipeline, in parallel with instruction fetch. Since the applications explored in this paper only require predictions of E nodes, only E nodes are sampled and predicted. It should be noted that D and C nodes are still required during training to accurately model the resource constraints of the critical path. We used a 16K entry array with 6-bit hysteresis, with a total size of 12 kilobytes.

The trainer is implemented as a small token array (Figure 6). The array stores information about the segment of the dependence graph for the *ROB_size* most recent instructions committed. One bit is stored for each node of these instructions, indicating whether the token was propagated into that node. Note that the array does not encode any dependence edges; their effect is implemented by the propagation step (see step 2 below). Finally, note that the reason why the token array does not need more than *ROB_size* entries is the observation that no critical-path dependence can span more than *ROB_size* instructions (see Section 2).

As each instruction commits, it is allocated an entry in the array, replacing the oldest instruction in a FIFO fashion. A token is planted into a node of the instruction by setting a bit in the newly allocated entry (**step 1** of Figure 5).

To perform the token propagation (**step 2**), the processor core provides, for each committing instruction, identification of the source nodes of the last-arriving edges targeting the three nodes of the committing instruction. For each source node, its entry in the token array is read (using its identification as the index) and then written into the target node in the

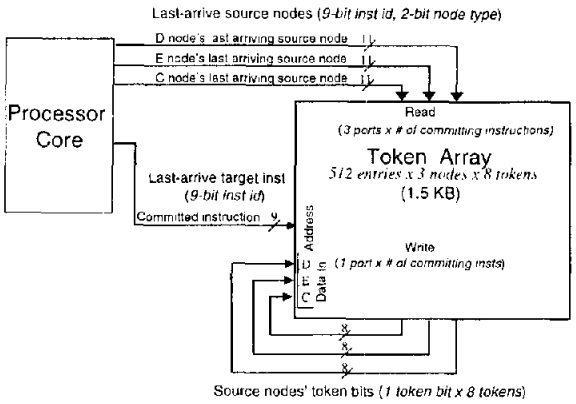


Figure 6: **Training path of the critical-path predictor.** Training the token-passing predictor involves reading and writing a small (1.5 kilobyte) array. The implementation shown permits the simultaneous propagation of 8 tokens.

committing instruction. This simple operation achieves the desired propagation effect.

Checking if the token is still alive (**step 3**) can be easily implemented without a scan of the array, by monitoring whether any instruction committed in the recent past has written (and therefore propagated) the token. If the token has not been propagated in the last *ROB_size* committed instructions, it can be deduced that none of the nodes in the token array holds the token, and, hence, the token is *not* alive. Finally, based on the result of the liveness check, the instruction where the token was planted is trained (**step 4**) by writing into the critical-path prediction table, using the hysteresis-based training rules in Table 3.

After the liveness check, the token is *freed* and can be re-planted (**step 1**) and propagated again. The token planting strategy is a design parameter that should be tuned to avoid repeatedly sampling some nodes while rarely sampling others. In our design, we chose to randomly re-plant the token in one of the next 10 instructions after it is freed.

There are many design parameters for the predictor, but, due to space considerations, we do not present a detailed study of the design space. The design parameters chosen for the experiments in this paper are shown in Table 3.

Discussion. Clearly, there is a tradeoff between the propagation distance and the frequency with which nodes can be sampled to check their criticality. If a token is propagating, it is in use and cannot be planted at a new node. If the propagation distance is too large, the adaptability of the predictor may be compromised. Nonetheless, a large propagation distance is desired for robust performance independent of the characteristics of particular workloads. We can compensate for this effect by adding hardware for multiple simultaneous in-flight tokens. These additional tokens are relatively inexpensive as all the tokens can be read and written together during propagation. For the propagation distance we chose (500 + ROB size = 1012 dynamic instructions), eight simultaneous in-flight tokens was sufficient. For this configuration, the token array size is 1.5 kilobytes (reorder buffer size $\times$ nodes $\times$ tokens = $512 \times 3 \times 8$ bits).

Critical path prediction table	12 kilobytes (16K entries * 6 bit hysteresis)
Token propagation Distance	1012 dynamic instructions (500 + ROB size)
Maximum number of Tokens in flight simultaneously	8
Hysteresis	Saturate at 63, increment by 8 when training critical, decrement by one when training non-critical. Instruction is predicted critical if hysteresis is above 8.
Planting Tokens	A Token is planted randomly in the next 10 instructions after it becomes available.

Table 3: **Configuration of token-passing predictor.**

Although the number of ports of the token array is proportional to the maximum commit bandwidth (as well as to the number of simultaneous last-arriving edges), due to its small size, the array may be feasible to implement using multi-ported cells and replication. Alternatively, it may be designed for the average bandwidth. Bursty periods could be handled by buffering or dropping the tokens.

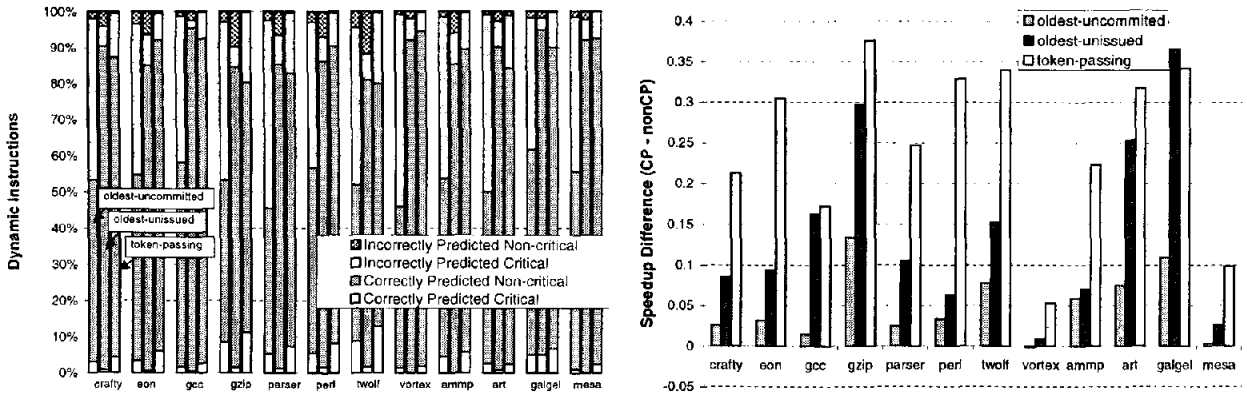
Notice in Table 3 that the hysteresis we used is biased to avoid rapid transition from a critical prediction to a non-critical prediction. The goal is to maintain the prediction for a critical instruction even after an optimization causes the instruction to become non-critical, so that the optimization continues to be applied. Together with retraining, the effect of this hysteresis is that *near-critical* instructions are predicted as critical after the critical instructions have been optimized.

Evaluation. Our token-passing predictor is designed using a *global* view of the critical path. An alternative is to use *local* heuristics that observe the machine and train an instruction as critical if it exhibits a potentially harmful behavior (*e.g.*, when it stalls the reorder buffer). A potential advantage of a heuristic-based predictor is that its implementation could be trivially simple.

Our evaluation suggest that heuristics are much less effective than a model-based predictor. We compare our predictor to two heuristic predictor designs of the style used in Tune, *et al.* [21]. The first predictor marks in each cycle the oldest *uncommitted* instruction as critical. The second predictor marks in each cycle the oldest *unissued* instruction if it is not ready to issue. We used the hysteresis strategy presented in their paper. Although our simulator parameters differ from theirs (see Section 4), a comparison to these heuristics will give an indication of the relative capabilities of the two predictor design styles.

We first compare the three predictors to the trace of the critical path computed by the simulator using our model from Section 2. The results, shown in Figure 7(a), show that we predict more than 80% of dynamic instructions correctly (both critical and non-critical) in all benchmarks (88% on average). Our predictor does a better job of correctly predicting *critical* instructions than either of the two heuristics-based predictors. Note that the *oldest-unissued* predictor has a relatively low misprediction rate, but tends to miss many critical instructions, which could significantly affect its optimization potential.

Second, to perform a comparison that is independent of our critical-path model, we study the effectiveness of the various predictors in an optimization. To this end, we performed the same experiment that we used for validating the critical-path model—extending all latencies by one cycle and then decreasing critical and non-critical latencies (see Figure 2(a) in Section 2). For an informative comparison, we plot the difference of the performance improvement from decreasing



(a) Comparison against ideal CP trace.

(b) Comparison via latency reduction.

Figure 7: **The token-passing predictor, based on the explicit model of the critical path, is very successful at identifying critical instructions.** (a) Comparison of the token-passing and two heuristics-based predictors to the "ideal" trace of the critical path, computed according to the model from Section 2. The token-passing predictor is over 80% (88% on average) accurate across all benchmarks and typically better than the heuristics, especially at correctly predicting nearly all critical instructions. (b) Plot of the difference of the performance improvement from decreasing critical latencies minus the improvement from decreasing non-critical latencies. Except for *galgel*, the token-passing predictor is clearly more effective.

critical latencies minus the improvement obtained when decreasing non-critical latencies. This yields a metric of how good the predictor is at identifying performance-critical instructions. The larger the difference, the better the predictions. The results are shown in Figure 7(b). The token-passing predictor typically outperforms either of the heuristics, often by a wide margin. Also, notice that the heuristics-based predictors are ineffective on some benchmarks, such as *oldest-uncommitted* on *gcc* and *mesa* and both *oldest-uncommitted* and *oldest-unissued* on *vortex*. While a heuristic could be devised to work well for one benchmark or even a set of benchmarks, explicitly modeling the critical path has the significant advantage of robust performance over a variety of workloads. Section 4 evaluates all three predictors in real applications. We will show that even the optimization being applied can render a heuristics-based predictor less effective, eliminating the small advantage *oldest-unissued* has for *galgel*.

4 Applications of the Critical Path

Our evaluation uses a next-generation dynamically-scheduled superscalar processor whose configuration is detailed in Table 5. Our simulator is built upon the SimpleScalar tool set [4]. Our benchmarks consist of eight SPEC2000 integer and four SPEC2000 floating point benchmarks; all are optimized Alpha binaries using reference inputs. Initialization phases were skipped, 100 million instructions were used to warm up the caches, and detailed simulation ran until 100 million instructions were committed. Baseline IPC and skip distances are shown in Table 4.

4.1 Focused cluster instruction scheduling and steering

Focused instruction scheduling and *steering* are optimizations that use the critical path to arbitrate access to contended resources (scheduling) and mitigate the effect of long latency inter-cluster communication (steering). Our experiments show that the two optimizations improve the performance of a next-generation clustered processor architecture by up to 21% (10% on average), with focused instruction scheduling providing the bulk of the benefit.

Benchmark	Base IPC	Insts Skipped (billions)
crafty (int)	3.75	4
eon (int)	3.50	2
gcc (int)	2.67	4
gzip (int)	3.03	4
parser (int)	1.63	2
perl (int)	2.61	4
twolf (int)	1.60	4
vortex (int)	4.65	8
ammp (fp)	3.14	8
art (fp)	2.10	8
galgel (fp)	4.19	4
mesa (fp)	5.04	8

Table 4: **Baseline IPCs and Skip Distances.**

The Problem. The complexity of implementing a large instruction window with a wide issue width has led to proposals of designs where the instruction window and functional units are partitioned, or *clustered* [2, 6, 10, 12, 13]. Clustering has already been used to partition the integer functional units of the Alpha 21264 [8]. Considering the trends of growing issue width and instruction windows, future high-performance processors will likely cluster both the instruction window and functional units.

Clustering introduces two primary performance challenges. The first is the *latency to bypass* a result from the output of a functional unit in one cluster to the input of a functional unit in a different cluster. This latency is likely to be increasingly significant as wire delays worsen [12]. If this latency occurs for an instruction on the critical path, it will add directly to execution time.

The second potential for performance loss is due to increased *functional unit contention*. Since each cluster has a smaller issue width, imperfect instruction load balancing can cause instructions to wait for a functional unit longer than in an unclustered design. If the instruction forced to wait is on

Dynamically Scheduled Core	256-entry instruction window, 512-entry re-order buffer 8-way issue, perfect memory disambiguation, fetch stops at second taken branch in a cycle.
Branch Prediction	Combined bimodal (8k entry)/gshare (8k entry) predictor with an 8k meta predictor, 2K entry 2-way associative BTB, 64-entry return address stack.
Memory System	64KB 2-way associative L1 instruction (1 cycle latency) and data (2 cycle latency) caches, shared 1 MB 4-way associative 10 cycle latency L2 cache, 100 cycle memory latency, 128-entry DTLB; 64-entry ITLB, 30 cycle TLB miss handling latency.
Functional Units (latency)	8 Integer ALUs (1), 4 Integer MULT/DIV (3/20), 4 Floating ALU (2), 4 Floating MULT/DIV (4/12), 4 LD/ST ports (2).
2 cluster organization	Each cluster has a 4-way issue 128-entry scheduling window, 4 Integer ALUs, 2 Integer MULT/DIV, 2 Floating ALU, 2 Floating MULT/DIV, 2 LD/ST ports.
4 cluster organization	Each cluster has a 2-way issue 64-entry scheduling window, 2 Integer ALUs, 1 Integer MULT/DIV, 1 Floating ALU, 1 Floating MULT/DIV, 1 LD/ST port.

Table 5: **Configuration of the simulated processor.**

the critical path, the contention will translate directly to an increase in execution time. Furthermore, steering policies have conflicting goals in that a scheme that provides good load balance may do a poor job at minimizing the effect of inter-cluster bypass latency.

The critical path can mitigate both of these performance problems. First, to reduce the effect of inter-cluster bypass latency, we perform *focused instruction steering*. The goal is to incur the inter-cluster bypass latency for non-critical instructions where performance is less likely to be impacted. The baseline instruction steering algorithm for our experiments is the *register-dependence* heuristic. This heuristic assigns an incoming instruction to the cluster that will produce one of its operands. If more than one cluster will produce an operand for the instruction (a *tie*), the producing cluster with the fewest instructions is chosen. If all producer instructions have finished execution, a load balancing policy is used where the incoming instruction is assigned to the cluster with the fewest instructions. This policy is similar to the scheme used by Palacharla *et al.* [12], but more effective than the dependence-based scheme studied by Baniasadi and Moshovos [2]. In the latter work, consumers are steered to the cluster of their producer until the producer has committed, even if it has finished execution. Thus, load balancing will be applied less often. Our *focused instruction steering* optimization modifies our baseline heuristic in how it handles ties: if a tied instruction is critical, it is placed into the cluster of its critical predecessor. This optimization was performed by Tune *et al.* [21].

Second, to reduce the effect of functional unit contention, we evaluated *focused instruction scheduling*, where critical instructions are scheduled for execution before non-critical instructions. The goal is to add contention only to non-critical instructions, since they are less likely to degrade performance. The oldest-first scheduling policy is used to prioritize among critical instructions, but our experiments found this policy does not have much impact due to the small number of critical instructions. The baseline instruction scheduling algorithm gives priority to *long latency* instructions. Our experiments found this heuristic performed slightly better than the *oldest-first* scheduling policy.

Experiments. The improvements due to *focused instruction scheduling* and *focused instruction steering* are shown in Figure 8(a) for three organizations of an 8-way issue machine: unclustered, two clusters, and four clusters (see Table 5). The execution time is normalized to the baseline machine (unclustered without any focused optimizations). We find that:

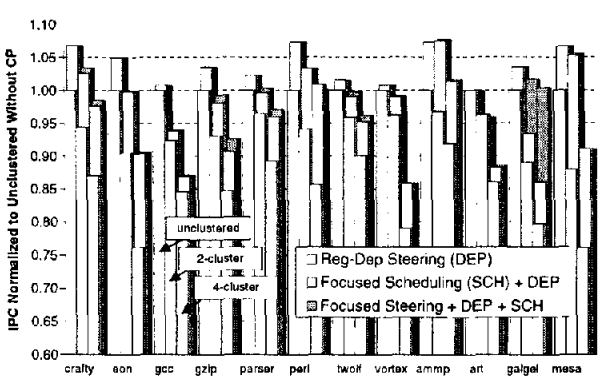
- On an unclustered organization, the critical path produces a speedup of as much as 7% (3.5% on average).
- On a 2-cluster organization, the critical path turns an average slowdown of 7% to a small *speedup* of 1% over the baseline. This is a speedup of up to 17% (7% on average) over register-dependence steering alone.
- On a 4-cluster organization, the critical path reduces performance degradation from 19% to a much more tolerable 6% degradation. Measured as speed up over register-dependence steering, we improve performance by up to 21% (10% on average).

From these results, we see that the token-passing predictor is increasingly effective as the number of clusters increases. This is an important result considering that technological trends may necessitate an aggressive next-generation microprocessor, such as the one we model, to be heavily partitioned in order to meet clock cycle goals [1].

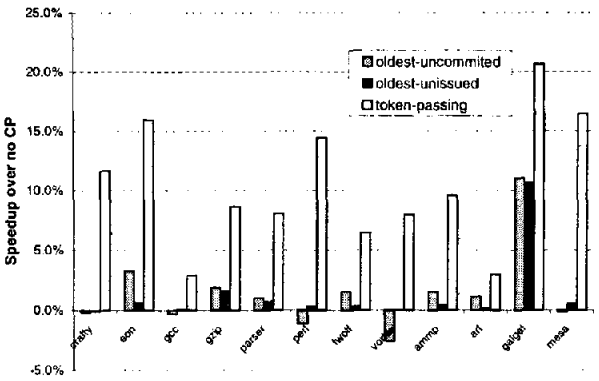
From Figure 8(a) we also see that *focused instruction scheduling* provides most of the benefit. We believe this is because *focused instruction steering* uses the critical path only to break ties, which occur in the register-dependence steering heuristic infrequently. Nonetheless, a few benchmarks do gain significantly from the enhanced steering, *e.g.*, *gzip* gains 3% and *galgel* gains 14%.

An alternative to *focused instruction scheduling* is to use a steering policy that prevents imbalance that might lead to excessive functional unit contention. We implemented several such policies, including the best performing non-adaptive heuristic (MOD3) studied by Baniasadi and Moshovos [2]. MOD3 allocates instructions to clusters in a round-robin fashion, three instructions at a time. While these schemes sometimes performed better than register-dependence steering, register-dependence performed better on average in our experiments. Most importantly, register-dependence steering with focused instruction scheduling *always* performed better (typically much better) than MOD3.

In Figure 8(b), we compare the token-passing predictor to the two heuristics-based predictors described in Section 3.2 (oldest-uncommitted and oldest-unissued) performing both *focused instruction scheduling* and *focused instruction steering* on a 4-cluster organization. Clearly, neither heuristics-based predictor is consistently effective, and they even degrade performance for some benchmarks (*e.g.*, for *vortex*, *perl*, and



(a) Scheduling in clustered architectures.



(b) Comparison to heuristics-based predictors.

Figure 8: **Critical path scheduling decreases the penalty of clustering.** **(a)** The token-passing predictor improves instruction scheduling in clustered architectures (8-way unclustered; two 4-way clusters; and four 2-way clusters are shown). As the number of clusters increases, critical-path scheduling becomes more effective. **(b)** Results for four 2-way clusters using both *focused instruction scheduling* and *steering* shows that the heuristic-based predictors are less effective than the token-passing predictor.

crafty). Our conjecture is that instruction scheduling optimizations require higher precision than heuristics can offer.

Note that even for *galgel*, where the oldest-unissued scheme compared favorably to the token-passing predictor in Section 3.2, Figure 7(b), the token-passing predictor produces a larger speedup. Upon further examination, we found that the oldest-unissued predictor's accuracy degrades significantly after *focused instruction scheduling* is applied. This may be due to the oldest-unissued predictor's inherent reliance on the order of instructions in the instruction window. Since scheduling critical instructions first changes the order of issue such that critical instructions are unlikely to be the oldest, the predictor's performance may degrade as the optimization is applied. In general, a predictor based on an explicit model of the critical path, rather than on an artifact of the microexecution, is less likely to experience this sort of interference with a particular optimization.

In summary, it is worth noting that the significant improvements seen for scheduling execution resources speak well for applying criticality to scheduling other scarce resources, such as ports on predictor structures or bus bandwidth. In general, the critical path can be used for intelligent resource arbitration whenever a resource is contended by multiple instructions. The multipurpose nature of a critical-path predictor can enable a large performance gain from the aggregate benefit of many such simple optimizations.

4.2 Focused value prediction

Focused value prediction is an optimization that uses the critical path for reducing the frequency of (costly) misspeculations while maintaining the benefits of useful predictions. By predicting only critical instructions, we improved performance by as much as 5%, due to removing nearly half of all value misspeculations.

The Problem. Value prediction is a technique for breaking data-flow dependences and thus also shortening the critical path of a program [11]. In fact, the optimization is only effective when the dependences are on the critical path. Any value prediction made for *non*-critical dependences will not improve performance; even worse, if such a prediction is incorrect, it may severely degrade performance. In *focused value prediction*, we only make predictions for critical path instructions, thus reducing the risk of misspeculation while maintaining the benefits of useful predictions.

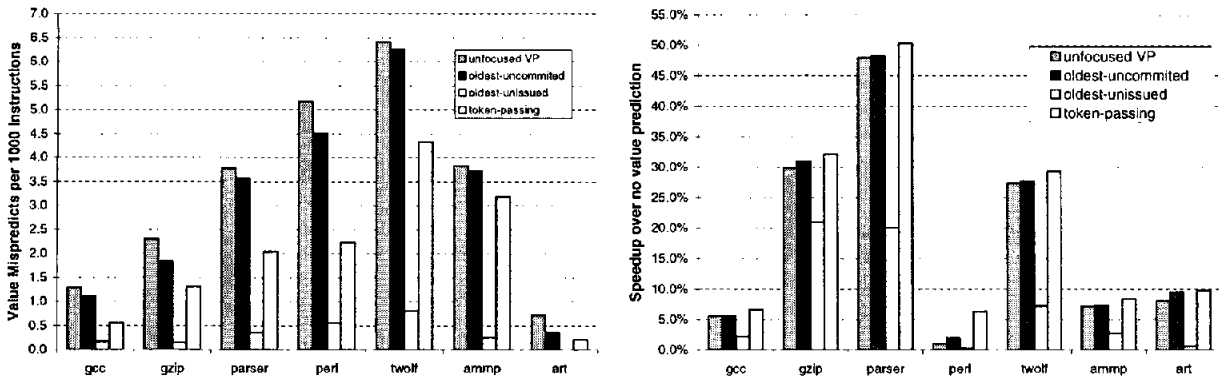
We could also use the critical path to make better use of prediction table entries and ports. However, because presenting all results of our value prediction study is beyond the scope of this paper, we restrict ourselves to the effects of reducing unnecessary value misspeculations. Our experiments should be viewed as a proof of the general concept that *critical-path-based speculation control* may improve any processor technique in which the cost of misspeculation may impair or outweigh the benefits of speculation, *e.g.*, issuing loads prior to unresolved stores.

Table Sizes	Context: 1st-level table: 64K entries, 2nd-level table: 64K entries, Stride: 64K entries. The tables form a hybrid predictor similar to the one in [22]
Confidence	4-bits, saturating: Increase by one if correct prediction, decrease by 7 if incorrect, perform speculation only if equal to 15 (This is similar to the mechanism used in [5]).
Mis-speculation Recovery	When an instruction is misspeculated, squash all instructions before it in the pipeline and re-fetch (like branch mispredictions.)

Table 6: **Value prediction configuration.**

Experiments. We used a hybrid context/stride predictor similar to the predictor of Wang and Franklin [22]. The value predictor configuration, detailed in Table 6, deserves two comments: In order to isolate the effect of value misspeculations from the effects of value-predictor aliasing, we used rather large value prediction tables. Second, while a more aggressive recovery mechanism than our squash-and-refetch policy might reduce the cost of misspeculations, it would also significantly increase the implementation cost. We performed experiments with focused value prediction on the seven benchmarks that our baseline value predictor could improve. We evaluate our token-passing predictor and the two heuristics predictors.

Figure 9(a) shows the number of misspeculations obtained with and without filtering predictions using the critical path. While the oldest-unissued heuristic eliminated the most misspeculations, it is clear from Figure 9(b) that it also eliminated



(a) Value misspeculations.

(b) Speedup of focused value prediction.

Figure 9: **Focusing value-prediction by removing misspeculations on non-critical instructions.** **(a)** A critical-path predictor can significantly reduce misspeculations. **(b)** For most benchmarks, the token-passing critical-path predictor delivers at least 3-times more improvement than either of the heuristics-based predictors.

many beneficial correct speculations. The more precise token-passing predictor consistently improves performance over the baseline value predictor and typically delivers more than 3-times more improvement than either heuristic. The absolute performance gain is modest because the powerful confidence mechanism in the baseline value predictor already filters out most of the misspeculations. Nonetheless, the potential for using the critical path to improve speculation techniques via misspeculation reduction is illustrated by 5 times more effective value prediction for *perl* and 7–20% more effectiveness for the rest of the benchmarks.

5 Related Work

Srinivasan and Lebeck [18] defined an alternative measure of the critical path, called latency tolerance, that provided non-trivial insights into the performance characteristics of the memory system. Their methodology illustrated how difficult it is to measure criticality even in a simulator wherein a complete execution trace is available. Their latency tolerance analysis involves rolling back the execution, artificially increasing the latency of a suspected non-critical load instruction, re-executing the program, and observing the impact of the increased latency. While their methodology yields a powerful analysis of memory accesses, their analysis cannot (easily) identify criticality of a broad class of microarchitectural resources, something that our model can achieve.

Concurrently with our work, Srinivasan, *et al.* [17] proposed a heuristics-based predictor of load criticality inspired by the above mentioned analysis. Their techniques consider a load as critical if (a) it feeds a mispredicted branch or another load that cache misses or (b) the number of independent instructions issued soon following the load is below a threshold. The authors perform experiments with critical-load victim caches and prefetching mechanisms, as well as measurements of the critical data working set. Their results suggest criticality-based techniques should not be used if they violate data locality. As the authors admit, there may be other ways for criticality to co-exist with locality. For example the critical-path could be used to schedule memory accesses.

Fisk and Bahar [7] explore a hardware approximation of the latency-tolerance analysis based on monitoring performance degradation on cache misses. If performance degrades below a threshold, the load is considered critical. They also look at heuristics based on the number of dependencies on a cache-missed load's dependence graph. While these heuristics provide some indication of criticality, our predictor is based on an explicit model of the critical path and hence is not optimization-specific: it works for all types of instructions, not just loads.

Tune *et al.* [21] identified the benefits of a critical path predictor and provided the first exploration of heuristics-based critical path predictors. We have thoroughly evaluated the most successful of their predictors in this paper. The evaluation led to a conjecture that critical-path-based optimizations require precision that heuristics cannot provide.

Calder *et al.* [5] guide value prediction by identifying the longest dependence chain in the instruction window, as an approximation of the critical path, without proposing a hardware implementation. We contribute a more precise model and an efficient predictor.

Tullsen and Calder [20] proposed a method for *software-based profiling* of the program's critical path. They identified the importance of microarchitectural characteristics for a more accurate computation of the true critical path and expressed some of them (such as branch mispredictions and instruction window stalls) in a dependence-graph model. We extend their model by separating different events in the instruction's lifetime, thus exposing more details of the microarchitectural critical path. Also, our model is well suited for an efficient hardware implementation.

6 Conclusions and Future Work

We have presented a dependence-graph-based model of the critical path of a microexecution. We have also described a critical-path predictor that makes use of the analytical power of the model. The predictor "analyzes" the dependence graph without building it, yielding an efficient hardware implementation. We have shown that the predictor supports *fine-grain* optimizations that require an accurate prediction of the critical path, and provides robust performance improvements over a variety of benchmarks. For instance, our critical-path predictor consistently improves cluster instruction scheduling and steering, by up to 21% (10% on average).

Future work includes refining the model and tuning the

predictor. While the precision of our current model is sufficient to achieve significant performance improvement, we believe higher precision would yield corresponding increases in benefit. For instance, in *focused value prediction*, if some truly critical instruction is not identified by the model, it will never be value predicted, even though the performance gain might be great. In a related vein, a more detailed study of the adaptability of the token-passing predictor during the course of optimizations might lead to a better design. It may be, for instance, that a different token planting strategy would be more effective for some optimizations. Maybe the predictor would adapt quicker if tokens were planted in the vicinity of a correct value prediction.

Another direction for future work is developing other critical-path-based optimizations. For instance, *focused resource arbitration* could be applied to scheduling memory accesses, bus transactions, or limited ports on a value predictor. *Focused misspeculation reduction* could be used to enhance other speculative mechanisms, such as load-store reordering or hit-miss speculation [23]. To conclude, the greatest practical advantage of the critical-path predictor is its multipurpose nature—the ability to enable a potentially large performance gain from the aggregate benefit of many simple optimizations, all driven by a single predictor.

Acknowledgements. We thank Adam Butts, Jason Cantin, Pacia Harper, Mark Hill, Nilofer Motiwala, Manoj Plakal, and Amir Roth for comments on drafts of the paper. This research was supported in part by an IBM Faculty Partnership Award and a grant from Microsoft Research. Brian Fields was partially supported by an NSF Graduate Research Fellowship and a University of Wisconsin-Madison Fellowship. Shai Rubin was partially supported by a Fulbright Doctoral Student Fellowship.

References

[1] V. Agarwal, M.S. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*, Vancouver, June 10–14 2000.

[2] A. Baniasadi and A. Moshovos. Instruction distribution heuristics for quad-cluster, dynamically-scheduled, superscalar processors. In *Proceedings of the 33th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-00)*, pages 337–347, December 10–13 2000.

[3] P. Barford and M. Crovella. Critical path analysis of TCP transactions. In *Proceedings of ACM SIGCOMM 2000*, January 2000.

[4] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-1997-1342, University of Wisconsin, Madison, June 1997.

[5] B. Calder, G. Reinman, and D. Tullsen. Selective value prediction. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*, pages 64–75, New York, N.Y., May 1–5 1999.

[6] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 149–159, Los Alamitos, December 1–3 1997.

[7] B. R. Fisk and R. I. Bahar. The non-critical buffer: Using load latency tolerance to improve data cache efficiency. In *IEEE International Conference on Computer Design*, Austin, TX, 1999.

[8] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10:9–15, October 1996.

[9] J. K. Hollingsworth. Critical path profiling of message passing and shared-memory programs. *IEEE Transactions on Parallel and Distributed Systems*, 9(10), October 1998.

[10] G. Kemp and M. Franklin. PEWs: A decentralized dynamic scheduling algorithm for ILP processing. In *International Conference on Parallel Processing*, pages 239–246, Aug 1996.

[11] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 226–237, Paris, France, December 2–4, 1996.

[12] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *24th Annual International Symposium on Computer Architecture*, pages 206–218, 1997.

[13] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 138–148, Los Alamitos, December 1–3 1997.

[14] A. Roth and G.S. Sohi. Speculative data-driven multithreading. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, Jan 2001.

[15] M. Schlansker, V. Kathail, and S. Anik. Height reduction of control recurrences for ILP processors. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 40–51, San Jose, California, November 30–December 2, 1994.

[16] M. Schlansker, S. Mahlke, and R. Johnson. Control CPR: A branch height reduction optimization for EPIC architectures. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 155–168, 1999.

[17] S. T. Srinivasan, R. Dz ching Ju, A. R. Lebeck, and C. Wilkerson. Locality vs. criticality. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, June 2001.

[18] S. T. Srinivasan and A. R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, pages 148–159, Los Alamitos, November 30–December 2 1998.

[19] J. Stark, P. Racunas, and Y. N. Patt. Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 34–45, Los Alamitos, December 1–3 1997.

[20] D. Tullsen and B. Calder. Computing along the critical path. Technical report, University of California, San Diego, Oct 1998.

[21] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, Jan 2001.

[22] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 281–291, Los Alamitos, December 1–3 1997.

[23] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculative techniques for improving load related instruction scheduling. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, June 1999.