The 11th International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2002.

Quantifying Instruction Criticality

Eric S. Tune Dean M. Tullsen Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
{etune, tullsen, calder}@cs.ucsd.edu

Abstract

Information about instruction criticality can be used to control the application of micro-architectural resources efficiently. To this end, several groups have proposed methods to predict critical instructions. This paper presents a framework that allows us to directly measure the criticality of individual dynamic instructions. This allows us to (1) measure the accuracy of proposed critical path predictors, (2) quantify the amount of slack present in non-critical instructions, and (3) provide a new metric, called tautness, which ranks critical instructions by their dominance on the critical path. This research investigates methods for improving critical path predictor accuracy and studies the distribution of slack and tautness in programs. It shows that instruction criticality changes dynamically, and that criticality history patterns can be used to significantly improve predictor accuracy.

1 Introduction

Critical path prediction [18, 7] classifies dynamic instructions based on their potential to affect performance. Using information about instruction criticality, limited processor resources can be used more efficiently. Specifically, value prediction, instruction steering, instruction scheduling, cache placement, and several power optimizations have been shown to benefit from criticality information.

This paper seeks to increase our understanding of the dynamic critical path and critical path predictors in several ways. Previous critical-path predictors produce a binary classification of criticality. Our research assigns a value to the criticality of instructions, which denotes the amount of benefit available from optimizing an instruction. We show the distribution of criticality and the variability of criticality for several programs.

Second, previous work predicts instruction criticality based on the program counter. However, our research shows that criticality is very dynamic for those instructions that actually are on the critical path at least some of the time; thus, PC-based predictors will have limited success. This paper examines whether instruction criticality is more highly correlated to other index functions which might include information such as criticality pattern history, branch history, or load history. In particular, we show that a critical path predictor which uses the local history pattern of criticality can significantly improve critical path prediction accuracy.

This paper presents a framework that identifies for each dynamic instruction both whether, and to what extent, it is critical. This technique is computationally intensive, and is not intended as another dynamic critical path predictor. Rather, it is a tool for understanding how dynamic instructions differ in their impact on program runtime.

In using this framework, our contributions are to (1) evaluate the quality of predictions for several proposed critical path predictors, (2) study the correlation between the criticality of dynamic instructions and the corresponding static instructions, (3) correlate criticality with other events (e.g., branch history and load history) in the pipeline, (4) measure the slack (distance from being critical) present in non-critical instructions, (5) present a definition of *tautness*, a quantification of the importance of critical instructions with respect to optimization, and (6) present the distribution of slack and tautness among instructions.

The rest of the paper is organized as follows: Section 2 summarizes prior work related to identifying critical instructions and exploiting that information. Section 3 describes the simulator and the benchmarks used in this study. Section 4 describes our approach to quantifying the criticality of instructions. Section 5 evaluates several different proposed critical path predictors. Section 6 studies the distribution of critical and non-critical instructions in programs. Section 7 concludes.

2 Critical Path Prediction and Related Work

Several previous papers have used the notion of instruction criticality to classify instructions for use in some optimization. These papers differ in what instructions are targeted (loads vs. all instructions), how critical instructions are identified or predicted, and how the criticality information is applied.

Srinivasan et al. study the latency-tolerance of loads in [16]. In their work, latency tolerance refers to the longest latency that a load instruction could have before impacting

performance. They find, for many loads, that the latency tolerance of a load does not match the level of the memory hierarchy where its data resides. Building on this observation, Fisk and Bahar [8] propose several methods for identifying critical loads, and propose placing data associated with non-critical loads in a small buffer to reduce contention in the primary cache. Srinivasan et al. [15] propose a method for identifying and predicting critical loads, and apply these predictions to two memory hierarchy optimizations. They study a victim cache that holds only lines touched by critical loads with the goal of reducing contention in the victim buffer. They also investigate a form of prefetching limited to data associated with critical loads with the aim of reducing bus contention. Racvik et al. [13] propose a method for identifying non-critical loads, which they term "non-vital loads." They propose a level 0 cache which holds only data associated with critical loads.

In Tune, et al., [18], we proposed several heuristic methods for predicting critical instructions. Certain events in the processor pipeline can suggest that an instruction is critical. While a heuristic does not guarantee that an instruction is critical, heuristic predictors have been shown to be useful in directing a number of micro-architectural optimizations. That paper examines the performance of the predictors for value prediction, where the number of value predictions per cycle is limited, and in which critical instructions are selected for value prediction in favor of non-critical instructions. It also examines the use of critical path prediction to steer instructions in a clustered architecture. Previously, Calder, et al., [2] proposed value predicting only instructions which were likely to be on the critical path. Value prediction [11, 9] is an optimization which speculatively removes data dependencies using predicted results.

Tune, et al [18] also introduces the critical path buffer (CPB) that allows the use of past criticality history to produce future predictions. The heuristic predictor of [18] is also used in Seng, et al. [14], and in a modified form, in [4].

Fields, et al [7] propose a different method for predicting critical instructions. They model program and processor constraints, including control dependencies, data dependencies, and a limited instruction window, using a directed graph. Their predictor uses tokens which are passed along the edges of this virtual graph to determine if an instruction is critical. They use critical path predictions to steer instructions to clusters and to select instructions for issue. They also study a value prediction scheme where only critical instructions are value predicted in order to reduce the number of mispredictions.

Casmira and Grunwald [3] propose a processor with slow and fast functional units. Instructions with sufficient slack would execute on slower functional units, thus saving power. Seng et al. [14] study an architecture which uses a critical path predictor to assign instructions to either slow or fast functional units and reduce power. Since non-critical instructions may be able to tolerate the slower functional units, performance loss is mitigated. They also study separate instruction queues for critical and non-critical instructions. They find that critical instructions (as identified by the predictor) only require an instruction queue with scalar, in-order issue, whereas non-critical instructions benefit greatly from an out-of-order instruction queue with multiple issue.

Fetch	2 basic blocks/cycle 8 instructions/cycle
Issue	8 instructions/cycle
Commit	8 instructions/cycle
Branch Predictor	8k/8k-entry local-history, 16k-entry global, 16k-entry choice 8-cycle mispredict penalty
L1 Data Cache L1 Inst Cache L2 Cache L3 Cache	16kB 2-way (8-cycle miss penalty) 16kB 2-way (8-cycle miss penalty) 256kB 4-way (20-cycle miss penalty) 1MB 4-way (100-cycle miss penalty)

Table 1. The processor parameters.

In [6], Fields et al. study the distribution of slack present in programs. They show how, on a clustered architecture with multi-speed functional units, they can measure whether instructions had sufficient slack to tolerate a slow resource by sending an instruction to a slow resource, and then measuring whether it became critical, according to the critical path predictor from [7].

In a different domain, Alexander et. al. [1] study the near-critical paths of the graphs of communication and computation in parallel programs. They propose a Maximum Benefit Metric that quantifies the maximum improvement in runtime possible by reducing the execution time of a section of code. The metric we propose quantifies the maximum benefit from removing the dependencies on a particular instruction.

3 Methodology

Our framework for this research consists of three parts: a detailed simulator that produces a dependency trace for each application, a directed graph of the dependencies in the program built from this trace, and a program called the *rescheduler* which computes the effect on the execution time of the program as various dependencies are changed. In this section, we describe the simulator and the set of benchmarks we use to generate the dependency trace and again to validate the results of the rescheduler. In the next section, we describe the rescheduler and the constraint-graph model.

Simulations are performed using a detailed architectural simulation of an out-of-order processor executing the Alpha instruction set architecture. Simulations for this research were performed with the SMTSIM simulator [17], used in single-thread mode. The simulated processor has a reorder buffer of 255 instructions. Our simulated processor does not have a limited instruction queue; it is only limited by the size of the reorder buffer. The processor can fetch, execute, and commit up to 8 instructions per cycle. It can fetch up to two non-contiguous basic blocks per cycle. The memory system models contention at each level of the memory hierarchy. The parameters for the processor are summarized in Table 3.

We chose 5 SpecFP2000 and 8 SpecINT2000 benchmarks for this study. The benchmarks were fast forwarded (emulated

Benchmark	Code	Input	Fast Forward $\times 10^6$
Floating Point			
ammp	amm	ref	2700
applu	apl	ref	500
equake	equ	ref	3000
galgel	gal	ref	2600
swim	swi	ref	800
Integer			
crafty	cra	ref	1000
eon	eok	ref (kajiya)	100
gap	gap	ref	1000
gcc	gc2	ref (200)	10
gzip	gzp	ref (program)	50
parser	par	ref	320
twolf	two	ref	2500
vpr	vpr	ref	1000

Table 2. The benchmarks used in this study.

but not simulated) a sufficient distance to bypass initialization and startup code before measured simulation began. Then, the cache and branch predictors were warmed up for 50 million instructions for all benchmarks. Finally, the critical path measurements are based on 10 million instruction-long traces after warmup. The benchmarks used, their inputs, and the number of instructions fast-forwarded, are shown in Table 2. The reference input was used for all benchmarks, and where there are multiple reference inputs, the one used is indicated.

4 Critical and Slackful Instructions

The notion of a critical path comes from operations research[10, 12], where a project is represented by a directed acyclic graph where nodes represent milestones and edges represent jobs to be done, and their dependencies. The *earliest finish time* of a project is the length of a longest path through that graph. The difference between the earliest time at which an activity could be started and the latest time at which it could be started without delaying the finish time, is termed the slack of that activity. More precisely it is termed the "total float"[12] (which Fields et al. [6] refer to as "global slack"). The edges with no slack are said to lie on the critical path. Delaying the initiation of any critical activity will delay the completion of the entire project.

A program executing on a processor can, to a significant extent, be modeled by such a graph. Many types of dependences and constraints can be modeled with graph edges. Fields et. al. propose such a model in [7], which is discussed below. The longest path length of this graph corresponds to the execution time of the program. They classify instructions as being "fetch critical", "execution critical", or "commit critical" if delaying the fetch, execution, or committing of the instruction would delay the overall program's execution. In this paper, we are only interested in whether instructions are "execution critical".

4.1 Slack and Tautness

This paper focuses on two metrics, slack and tautness, to quantify instruction criticality. Intuitively, slack represents how far an instruction is from becoming critical. *The slack of an instruction is the number of cycles that the instruction can be delayed without increasing the execution time of the program.* Instructions with more than zero cycles of slack are non-critical.

We propose a new metric, *tautness*, for distinguishing critical instructions, which corresponds to how far away an instruction is from becoming non-critical. Tautness is a complementary measurement to slack, for instructions which are critical. *We define the tautness for an instruction as the number of cycles by which execution time is reduced when the result of that instruction was made available to other instructions immediately.* For all instructions that write a result to a register, this means making that result available as soon as the producing instruction is dispatched. For store instructions, this includes making the value stored available to dependent loads. For mispredicted branches, this means removing the misprediction.

Tautness is a useful measurement because it quantifies the maximum benefit of applying an optimization to an instruction. It roughly models what might be achieved by value predicting or speculatively precomputing the result of an instruction. Notice that this is not information that is available from identifying and analyzing the longest path through the graph. For example, an instruction with a latency of 100 cycles would thus contribute 100 cycles to the length of the longest path, but removing that instruction from the program graph might expose another path whose total length is only 1 cycle shorter than the original path. In that case, the instruction has a tautness of 1 cycle. Tautness accounts for all paths through the program, not just the longest.

The design of an implementable critical path predictor that returns a tautness value is left to future work, but such a predictor would have several benefits. (1) If the critical path predictor is used to arbitrate a constrained resource, a binary critical path predictor cannot distinguish between multiple critical instructions which want to use the resource. (2) Some optimizations, such as speculative precomputation [5, 4], devote significant resources to target a single instruction. In those cases, it is not sufficient to target critical instructions, but rather we would only want to target critical-path instructions that exceeded a tautness threshold. Speculative precomputation [5] could use a static critical path predictor that included tautness (that might look something like our rescheduler), but dynamic speculative precomputation [4] would require a dynamic predictor.

Our critical-path analysis framework allows us to precisely measure these two properties of instructions (slack and tautness). We first discuss the constraint-graph model of the critical path that we base our work on, and the algorithms we used to extend the constraint-graph model and to compute slack and tautness.

4.2 The Constraint Graph

We start with the graph model presented by Fields, et al. as a model of the critical path on a microprocessor. Their model includes not just the data dependencies of instructions, but also dependencies corresponding to control and some resource constraints, such as a finite instruction window. Another key feature is that each instruction is represented by several nodes, corresponding to different events as the instruction moves through the pipeline.

In the constraint graph, nodes represent an instruction reaching a particular pipeline stage in the machine, and edges represent latencies between those nodes. There are three nodes in the graph for each instruction, representing the time when the corresponding instruction is dispatched (*d-nodes*), executed (*e-nodes*), and committed (*c-nodes*). Those stages are always connected for a single instruction.

The graph is built from a trace of the execution generated by a detailed simulator. Edges between different types of nodes correspond to different hazards in the processor. Each edge has a weight, in cycles. For example, the following edges between different instructions are possible:

D $\rightarrow$ D edges arise between consecutive instructions, and indicate that instructions are constrained to be fetched serially. This edge has a weight only if there is a reason the instructions cannot be fetched the same cycle.

E $\rightarrow$ D edges indicated a control dependence caused by a branch misprediction. E $\rightarrow$ E edges represent a data dependence between instructions, and the weight is the latency of the producer.

C $\rightarrow$ D edges arise between an instruction and an instruction that is fetched 256 (in this case) instructions later, indicating that with a finite-size (256-instruction) reorder buffer, the first instruction must retire before the other can enter the buffer.

More details of the graph can be found in [7].

4.3 The Rescheduler

In [7], Fields et al. used the graph model to determine what instructions were critical in a particular execution of a program. In this paper, we use the graph-model to efficiently determine what would be the impact on the execution time of the program if each instruction were executed sooner or later. We use a program which we call the *rescheduler* to efficiently determine the effects of changes to a large graph. We also use the rescheduler to model a processor constraint, limited issue bandwidth, which cannot be represented in the graph.

The simulator which is used to generate program traces is described in Section 3. A program trace provides information about each dynamic instruction, including fetch delays, execution latency, execution dependencies, and branch mispredictions. Using the rescheduler, which takes this trace as input, and converts it into the directed graph model of [7], we can compute the effect of making an instruction complete execution earlier or later than it did in the original simulation. The rescheduler can compute the effect of changing a dependence faster than rerunning a simulation, and thus it is practical for us to make separate measurement for each instruction in a program trace containing tens of millions of instructions.

4.4 Rescheduler to Measure Slack and Tautness

We use the rescheduler to compute two metrics for each instruction: *slack* and *tautness*, as defined earlier. The rescheduler operates on a moving window of the graph, since the entire graph would be too large to store efficiently in memory. The longest path to each node is computed for all nodes, which are already in topologically sorted order as generated from the initial simulator trace. Next, to compute the effect of a change in the graph, the graph is changed as desired, and the longest path is recomputed for all nodes that follow the changes. However, it is not necessary to recompute the longest path for the entire graph each time a node is changed in order to determine the total effect on the program execution time. We exploit the fact that no edge spanning more than R instructions is ever on the longest path, where R is the size of the instruction window. As the longest path to each node is recomputed, the change in the time between the original schedule and the modified schedule is computed. When this difference, Δ, is constant for a run of consecutive instructions of length R, then we can say with certainty that all subsequent nodes in the graph will also change by Δ; thus, the runtime of the program changes by Δ. The rescheduler is feasible, even as an offline technique, only because permutations of the constraint graph always have a localized effect on the entire graph.

To compute the tautness for an instruction I, the rescheduler removes any data-dependence edges out of I's E-node. This allows instructions that depend on the result of I to execute independently of I. However, I must still execute eventually. Thus, if I is "commit critical" – it causes the instruction window to fill up when a critical instruction is just outside the window – then I will have a tautness of 0. We chose to define tautness this way so as to be similar to optimizations such as value prediction, where the instruction that is the target of optimization must still execute to validate speculative data.

To compute the slack for an instruction I, we delay the execution of I by a large number of cycles, and recompute the longest path for the graph. The difference between the amount by which I was delayed and the increase in the longest path is the slack in executing instruction I. For example, if delaying instruction I by 100 cycles causes the program to run 98 cycles longer, we conclude that I has 2 cycles of slack, after which it became critical.

Using this graph-adjustment approach, we compute the slack and the tautness for each e-node in the graph (every dynamic instruction in the program trace.)

We also augmented the longest path computation to adjust for the effect of a finite number of functional units. For the execution-node, e, of instruction I, the longest path to node

e would correspond to the cycle at which I executes, *if* there was not an issue limit. To model this additional constraint, after computing the longest path $l(e)$ to a node e, we consult a table to see how many older instructions were scheduled for the same cycle. If all functional units are already busy at cycle $l(e)$, then we increase $l(e)$ until a issue slot is found. This works because we assume that the hardware scheduler gives preference to older instructions when picking from among all of the available instructions that are ready to execute. This is only one of the resource constraints previous graph-based approaches do not handle well, but it serves as an example of how others could be handled, such as more specific functional unit constraints (load-store units, dividers) or a limited size instruction queue. These limitations are handled by a combination of the constraint graph and a mechanism for processing the graph.

In [7], the cycles that an instruction spends ready and waiting in the queue due to functional unit contention are included in the weight of the EE and EC edges. This is sufficient for determining whether an instruction was critical in the unchanged graph. When we model the token passing predictor of [7] in this paper, we include contention cycles in this fashion. But that is not adequate for our purposes, since we are interested in the effect of changing the graph, which also changes the conflict patterns. Therefore, we ignore any contention present in the program trace, and compute the effect of contention in the rescheduler.

It should be noted that slack and tautness are not completely mutually exclusive. Instructions along two paths of the same length will exhibit no tautness and no slack. But also, instructions that are not otherwise critical but use an issue slot that would have gone to a critical instruction, can exhibit both slack and tautness. This is because either accelerating the instruction or delaying the instruction can improve execution time.

4.5 Validation of the Rescheduler

The rescheduler and the dependence graph together incorporate many but not all of the effects modeled by a full simulation. Both the rescheduler and the simulator compute the cycle when each instruction is executed, and the total number of cycles to execute the program. There is a certain amount of error in the cycle times computed by the rescheduler due to effects not modeled.

In order to validate using the rescheduler/graph to calculate slack and tautness, we randomly selected dynamic instructions with a range of different slack and tautness values, and then compare the slack or tautness computed by the rescheduler/graph with their corresponding values from the detailed simulator. To measure slack and tautness in the simulator, we ran the simulator with that one dynamic instance of the instruction delayed (to measure slack) or with that one instruction's result available early (to measure tautness).

The left graph in Figure 1 is a scatter plot showing the agreement between the slack measured by the simulator and the slack measured by the rescheduler, for a random selection of instructions that our technique indicates has slack. The right graph in Figure 1 shows the same type of scatter plot, but for tautness.

We validated the rescheduler on a range of benchmarks. We present the results here for `twolf`, because those results fell in the middle of the benchmarks measured — some correlated better, some worse. The line $x = y$ is drawn for convenience. Points that lie on this line represent instructions where the rescheduler agreed exactly with the results of resimulation. For most instructions, the tautness (or slack) measured by the rescheduler is at or very close to the result obtained from a full detailed simulation run. The next section discusses some of the reasons why the correlation is not perfect.

4.5.1 Sources of Error in the Model

We use the graph/rescheduler to measure slack and tautness faster than would be possible through resimulation. The constraint graph is, however, a simplification of all the interactions that take place in a real processor, and some inaccuracies will result.

One type of discrepancy occurs because the memory hierarchy is only modeled indirectly in the graph. Load instructions have a variable latency. The latency on data-dependence edges associated with a load instruction are the actual execution latency of the load during the initial simulation. Once the weights are assigned, they do not change. In the most common case, the execution latency of a load instruction is the same regardless of changes to the graph. That is, in the common case, the latency of a load is independent of when it, and other instructions, execute.

However, there are three ways in which memory latencies can change that are not modeled by the rescheduler. First, the simulator models conflicts between non-dependent instructions for cache banks and data buses, but the rescheduler does not. Second, changes to load ordering can create (or eliminate) new cache conflict misses that the rescheduler will not recognize. Third, and we found this to be more significant than the first two, there is an indirect dependence between loads that access the same cache line. The first load to access the line essentially does a full or partial prefetch of the line for the other loads. If the loads are reordered, a different load sees the full latency of the access, and the original first load no longer does. These types of error also affect the slack measurements of [6].

5 Comparing Critical Path Predictors

Using the framework described in the previous section, we can know whether delaying each dynamic instruction is safe, and likewise, whether optimizing each dynamic instruction is worthwhile. We use this information about instructions to evaluate the usefulness of several proposed critical path predictors. We also explore the potential for new types of critical path predictors.

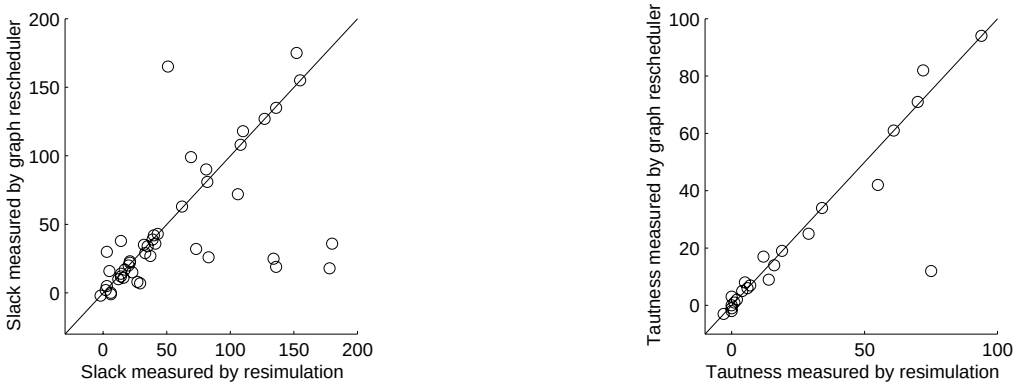

Figure 1. Scatter plot comparing the slack of instructions in the left graph and tautness of instructions in the right graph as computed by the rescheduler and as determined by re-simulation, for `twolf`, a representative benchmark.

Critical path prediction differs from other types of prediction, such as branch prediction, in an important respect. In branch prediction, a predictor table is trained using the outcomes of each branch, and there is no ambiguity over what this training information is. In critical path prediction, there is also a predictor table, trained based on the criticality of each instruction, but identifying whether an instruction was critical is a large part of the challenge. Thus, the accuracy of a critical path prediction depends on both the accuracy of the training stream (identifying whether an instruction was critical after it executed) and the accuracy of predictions (how training information is used by the prediction table to predict future criticality). We study both the *identification accuracy* of different proposed methods, and the *prediction accuracy* when a perfect identification method is coupled with different prediction tables.

5.1 Training Accuracy

This section examines the accuracy of the training mechanism used by several critical path predictors. We define an instruction as being non-critical if it has more than 0 cycles of slack, as measured by the rescheduler, or critical otherwise. Figure 2 shows the breakdown of correct and incorrect identification for various methods of identifying critical instructions across the 13 benchmarks. Each group of bars represents a benchmark. Within each group, individual bars represent different methods of identifying critical instructions. A letter at the top of each bar indicates the identification method. In this figure, *QOld* and *ALOld* represent critical path predictors from [18] using the QOld heuristic and the ALOld heuristic, respectively. QOld identifies instructions that become the oldest in the instruction queue, and ALOld identifies instructions that become the oldest in the active list (oldest non-retired). *Token* is the critical path predictor from [7], which plants a token at an instruction, and observes whether the token stays alive, propagating between instructions along last-arriving dependence edges. We modeled a token-passing predictor which could train 8 instructions at a time, with a 500-instruction training distance. *NonVital* is the load-criticality predictor from [13]. The NonVital predictor marks a load as "vital" if its result is used immediately.

The number of identifications made by each method varies. Thus, the ratio of the number instructions which are actually critical versus non-critical need not be the same for all bars in a group. The Heuristic methods (Q,A) make an identification for every instruction. The Token-passing predictor (T) only identifies a sample set of all instructions. The non-vital predictor only predicts loads. The exact fraction of identified instructions that are critical depends on the implementation of the predictor. For instance, in `ammp`, nearly half of all load instructions are critical, and in `galgel`, very few load instructions are critical, hence the fraction of critical instructions is clearly different for the NonVital bar as compared to the other predictors.

Figure 2 shows that the token-passing method does very well overall. It does particularly well at correctly identifying instructions that are actually non-critical – the heuristic techniques tend to be much more liberal in identifying potential critical instructions (intentionally so, hoping to capture instructions that are only occasionally critical [18]). For many optimizations, however, the most important category can be the mis-identification of critical instructions. For example, in a processor with clustered functional units, mistakenly sending a critical instruction to a different cluster from other critical instructions will have a direct cost: increased execution time due to bypass penalty. But mistakenly sending a non-critical instruction to the wrong cluster only has an indirect cost: possibly causing contention. The token-passer also does well with this criteria overall, but in several cases does not have the highest coverage of critical instructions.

The Non-Vital predictor (V) was proposed only as a predictor for Load instructions. The results show that the Non-Vital Loads technique does especially well at identifying critical

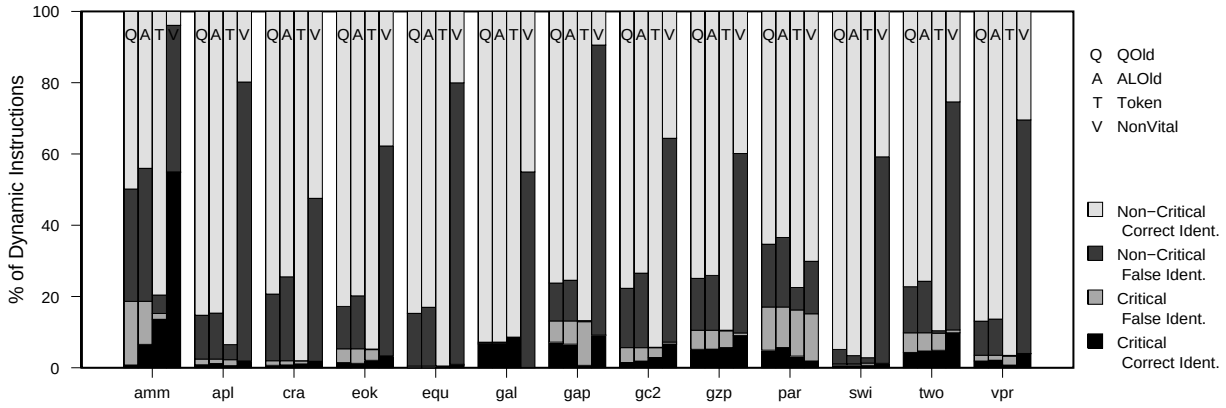

Figure 2. Correct and Incorrect Identification for 4 Different Criticality-Identification Mechanisms.

	>99%	>95%	>90%	>50%	>10%	>1%	>0%
amm	3.38	3.64	3.64	6.59	13.69	18.63	20.36
apl	6.66	6.66	6.67	7.39	15.83	19.30	22.49
cra	0.67	0.72	0.78	2.20	11.86	25.70	34.68
eok	0.45	0.51	0.58	1.15	6.58	12.03	18.02
equ	0.00	0.00	0.00	0.00	0.85	2.55	3.77
gal	0.00	2.18	4.37	4.37	4.80	16.16	38.86
gap	0.22	0.48	0.63	2.82	14.48	23.35	30.87
gc2	2.59	2.62	2.65	3.79	15.61	29.96	34.49
gzp	3.34	3.43	4.34	6.90	21.18	27.13	38.59
par	3.29	3.46	3.70	7.27	20.95	32.56	37.15
swi	0.00	0.00	0.00	0.00	11.60	19.89	25.97
two	0.40	0.40	1.03	4.53	20.79	35.31	51.93
vpr	0.14	0.14	0.14	0.55	6.57	10.82	17.36
AVG	1.48	1.71	2.07	3.41	12.59	21.23	29.52

Table 3. Criticality Bias of Static Instructions.

loads, but it does poorly at correctly identifying non-critical loads.

Note that the Token-passing predictor rarely identifies non-critical instructions as critical, because its prediction is based on a graph the same graph we use to measure criticality. But because we define criticality based on the slack measured using the rescheduler, which incorporates an additional processor constraint (limited issue width), such mispredictions are possible.

5.2 Criticality Bias of Static Instructions

Given a means to accurately identify which instructions are critical, the critical path predictor then uses that information to produce a prediction for future instructions.

If criticality is highly biased for individual static instructions, a simple predictor, even a static predictor, would be sufficient. This section examines the criticality bias of static instructions for a particular processor configuration. The results are shown in Table 3. In that table, a column labeled x% shows, for each benchmark, the fraction of static instructions for which more than x% of its dynamic instances were critical.

Looking at the last column of the table, between 4% and 52%, and on average 30% of static instructions are critical at least once. Thus, on average, 70% of static instructions can safely be ruled out as not being critical. This suggests that techniques that need to find a large number of, but not necessarily all, instructions with slack may not require a complex predictor.

Among static instructions which are at any point in time critical, those instructions tend to change their criticality often. For three of the benchmarks (equake, swim, and vpr), less than 1% of static instructions are critical even half of the times they are executed. That is, there are virtually no "statically critical" instructions for those benchmarks.

If we want to try to predict the critical path of the program with high accuracy, then a purely PC based approach will not be sufficient. According to the table, on average over all benchmarks, 33% of static instructions are critical at least once (column labeled 0%) and 12% of static instructions are critical more than 50% of the time. Thus, 21% are critical, but at a frequency less than or equal to 50%.

Table 4 shows the distribution of static instructions according to how often they change criticality (from critical on one dynamic instance to non-critical the next, or vice versa.) In this table, a column labeled $P > x$, with value y for some benchmark, means that y% of static instructions have a probability greater than x of changing their criticality between any two subsequent instances. For example, a static instruction that was critical 50 times in a row, and then non-critical 50 times in a row, and so on, would have $P = 2\%$. A different static instruction that alternates between critical and non-critical every time would have $P = 100\%$.

This table shows that on average 23% of static instructions tend to change their criticality more often than every 100 instances. Thus, a predictor which identifies the criticality of a static instruction, say, every 10 instances, would be able to predict 75% of static instructions with reasonable accuracy. This bodes fairly well for predictors like the token-passing predictor that produce intermittent training information. However, predictors that produce more frequent training information, like the Heuristic predictor, may still be able to use that to an advantage. 14% of static instructions overall (and up to about 25% for some benchmarks) change criticality at every 10 invocations or more. For example, consider a load instruc-

Benchmark	$P > 0.9$	$P > 0.5$	$P > 0.1$	$P > 0.01$
amm	0.35	1.91	12.74	18.20
apl	7.73	8.83	17.05	19.61
cra	0.53	1.50	15.05	28.53
eok	0.10	0.79	7.44	13.32
equ	0.00	0.00	1.60	2.55
gal	0.00	0.00	0.44	19.21
gap	0.78	2.88	16.65	24.93
gc2	2.48	3.78	18.80	32.03
gzp	4.30	4.90	22.27	29.08
par	2.82	5.00	23.82	34.03
swi	0.00	0.00	14.92	21.55
two	0.00	1.60	26.25	39.21
vpr	0.08	1.40	8.00	11.77
AVG	1.57	2.56	14.36	22.99

Table 4. Fraction of Static Instructions which Change Criticality with Different Frequency.

tion that has a cache miss every 8th time, because it reads 8 sequential words from a cache line. That instruction or its dependents might be non-critical 7 times and then critical 1 time, and repeating in that pattern. Predicting criticality based on a saturating counter would not be effective for this instruction. The next section examines the possibility of identifying patterns and correlations which can increase the predictability of the dynamically critical instructions.

5.3 Prediction

Previous work in critical path prediction used a PC-indexed prediction table with biased counters. This means that the address of a an instruction is used to index into a table of saturating counters, like a branch predictor, and that the counters are incremented by a large amount when an instruction is identified as critical, and decremented by 1 when an instruction is found to be non-critical. Both the token-passing predictor and the heuristic predictor used such a prediction table. We examine the accuracy that can be obtained with different types of prediction tables. In this section, we are less concerned with the practicality of implementing a prediction table, and more interested in the limits of how well critical instructions can be predicted. Therefore, we use Oracle training for all the predictors in this section.

Figure 3 shows the accuracy of different types of prediction tables. Each group of bars represents a benchmark. Within each group, individual bars represent different predictors. A code at the top of each bar indicates the predictor. The predictors are as follows: **One-Level** (**1** in the figure) – A one-level predictor consisting of a PC indexed table of 2-bit saturating counters. The table is unaliased. **1-Level, Biased Counter (X)** – A one-level predictor consisting of a PC indexed table of 5-bit saturating counters. The biased counter increments by 8 when an instruction is identified as critical, and decrements by 1 otherwise. **Two-Level (2)** – A two-level predictor, PC indexed table of 8-bit local histories (unaliased). Local history is used to index into a table of 256 2-bit saturating counters. **Branch History (B)** – A 1-level predictor, indexed by a concatenation of PC and 8 bits of branch-direction history, unaliased. **Branch Miss History (M)** – A 1-level predictor, indexed by a concatenation of PC and 8 bits of branch-misprediction history, unaliased. **Load Miss History (L)** – A 1-level predictor, indexed by a concatenation of PC and 8 bits of load-miss history, unaliased. All predictors use oracle training. Note that there are two subplots with different vertical scales, to show detail on those benchmarks that have a small percentage of critical instructions.

Each of the history-based predictors (BranchHistory, BranchMissHistory, LoadMissHistory) use 8 bits of history regarding the last 8 branch or load instructions. BranchHistory refers to the direction of previous branches. BranchMissHistory refers to whether previous branches were mispredicted. LoadMissHistory refers to whether previous loads were cache misses. Previous means older instructions, in program order. Information about the current instruction is not incorporated in the history.

Branch history would help prediction if the criticality of an instruction is highly correlated with the control flow path the program took to get to it. Branch history (B) is sometimes better than a simple PC-indexed prediction (1), and sometimes worse. The criticality of instructions can be affected by nearby cache misses and branch mispredictions, but these patterns did little to improve prediction accuracy except in the isolated case of `swim`. Note that miss history and mispredict history are hard to gather in real time, but our focus was on understanding the causes of varying criticality.

These results reinforce the findings in the previous subsection; Most static instructions are always not-critical, and so all of the predictors are able to identify nearly all non-critical instructions. The critical path runs through a small set of static instructions, but which of those are critical can vary frequently. Thus, if we are willing to tolerate predicting some non-critical instructions as critical (accept low accuracy to get high coverage of critical instructions), then the best approach is to predict as critical any static instruction that was recently critical. This is highlighted by the "Critical/Correctly Identified" bar for "Perfect-OneLev-Biased", which is always very tall. Thus the approach taken by two critical path predictors [18, 7] of biasing the counters (by incrementing by a large amount for critical, and decrementing by a small amount for non-critical) is effective.

However, if we are not willing to sacrifice accuracy, the two-level predictor had significantly higher coverage of critical instructions among the those that were not biased (few non-critical instructions called critical). In several cases, it achieves twice the coverage of the PC-based 1-level predictor. This indicates that many instructions do indeed follow a predictable pattern of criticality that can be identified by a local history predictor.

However, the two-level predictor would not be compatible with a sampling-based identification method, such as the token-passing predictor. Recall that the token-passing method does not produce training information for every instruction, but has a better training accuracy than the heuristic methods, which do sample every instruction. Although tokens can be placed in a controlled fashion to profile several consecutive instances of a static instruction, this would need to be continued

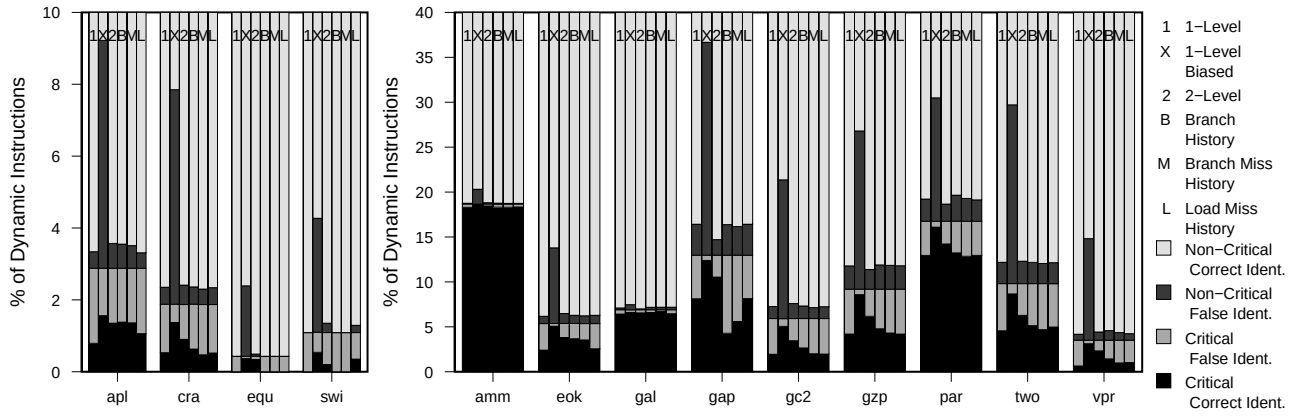

Figure 3. Correct and Incorrect Predictions for Different Prediction Tables, using Oracle Criticality ID.

indefinitely to maintain the local history for an instruction.

This section has demonstrated that predictors that predict based on PC and are slow to change predictions have a difficult time predicting critical instructions accurately, due to the highly dynamic behavior of those instructions. It has shown the potential for a pattern-based predictor to be more effective. The patterns by which instructions change their criticality warrants further study. One likely source of predictable patterns of criticality could stem from loads that access memory sequentially, missing for the first access to a line, and then hitting on subsequent accesses to that line.

6 Distribution of Critical Instructions

Slack and tautness are two metrics that provide more fine-grained information about criticality than a binary prediction (critical vs. non-critical). This section confirms this by showing slack and tautness are both highly varied in the set of programs we are considering.

The graphs in Figure 4 show the cumulative distribution of tautness values for dynamic instructions in the floating-point and integer benchmarks. A point on the curve shows what percentage of instructions have at least a certain number of cycles of tautness. Where a curve intersects the y-axis indicates the percentage of dynamic instructions that are critical.

Most integer benchmarks (the graph on the right) have fewer than 2.5% of instructions with tautness of more than 10 cycles. The benchmarks twolf and parser stand out as exceptions. Three of the 5 floating-point benchmarks have fewer than 1% of instructions with tautness of more than 10 cycles. We attribute the reduced amount of tautness in some floating point programs in part to loop unrolling and instruction scheduling, which will increase the number of similar-length, independent dependence chains, so that optimizing just one instruction (as tautness measures) will leave several other equally long, parallel chains which become critical. Since we define tautness in terms of making the result of an instruction available as soon as it is dispatched, rather than just reducing its latency, it is possible for an instruction to have a tautness much greater than the longest latency of any instruction in our simulator (about 360 cycles). This is most evident in `ammp`, which has a significant number of instructions with a tautness greater than 2000 cycles. Several of the integer programs also have a significant number of instructions with hundreds of cycles of tautness, as indicated by the long tails on the curves.

Figure 5 show histograms of the amount of slack in dynamic instructions. Notice that, particularly for the integer programs, there is a correlation between programs with high tautness values and high slack values. The average number of instructions with slack is much higher than the number of critical instructions, which was one of the original motivations for critical path prediction.

7 Conclusions

This paper describes a framework for measuring the degree of criticality (tautness) or non-criticality (slack) of each instruction in a program. Tautness is a new metric we introduce to measure the degree of criticality of a dynamic instruction. We evaluate the accuracy of several critical path predictors, both in terms of the accuracy of the training stream and the prediction mechanism, as well as demonstrate the potential for new prediction techniques. We find that there are more slackful instructions than taut instructions, and examine this distribution of slack and tautness in programs.

We found that a majority of static instructions are never critical, but among those static instructions that are ever critical, criticality varied frequently – very few static instructions are always critical; about 1.5% on average over 13 benchmarks. Thus, predicting exactly the dynamic instances of these static instructions that are critical is difficult, but important for a highly accurate predictor. We show that new prediction techniques that recognize patterns in these critical instructions have the potential to significantly increase the accuracy of critical path predictors.

This work suggests several important future directions to improve the effectiveness of critical path prediction. It shows that critical path predictors must be able to identify patterns

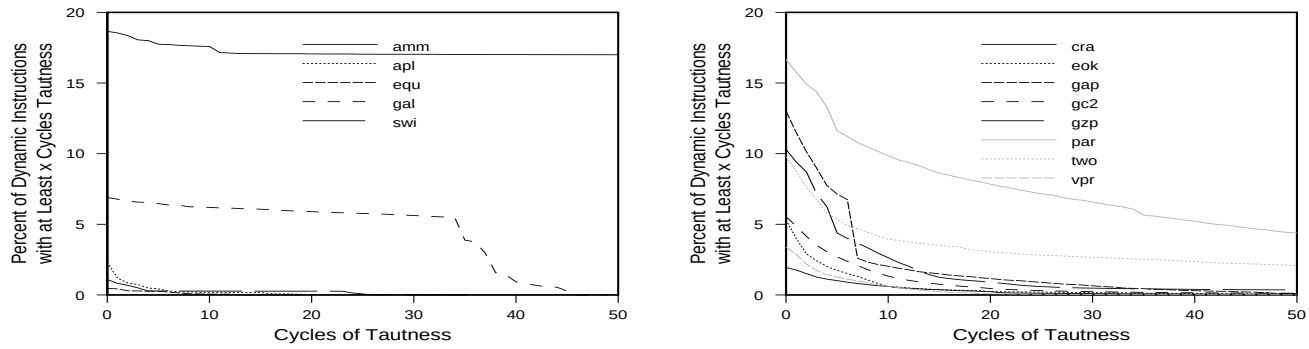

Figure 4. Cumulative Distribution (decreasing) of the Fraction of Dynamic Instructions with Tautness.

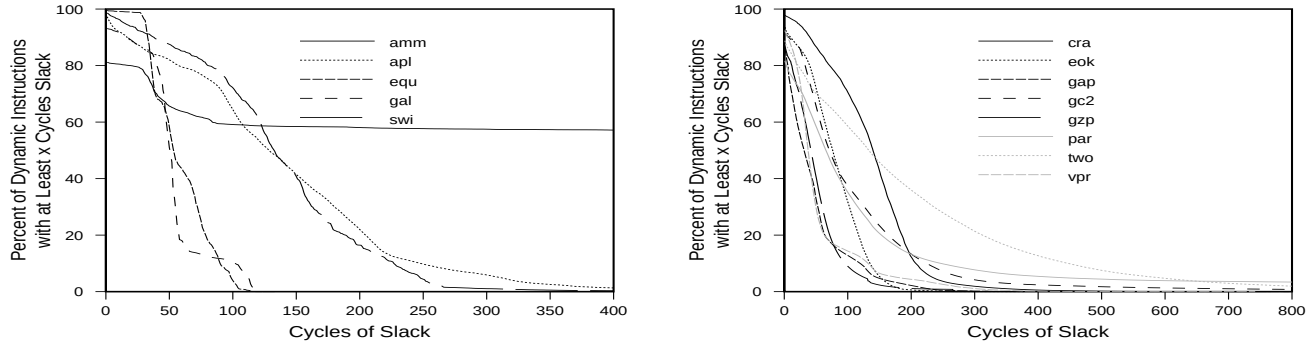

Figure 5. Cumulative Distribution (decreasing) of the Fraction of Dynamic Instructions with Slack.

of criticality to achieve high coverage and accuracy. It also demonstrates the need for predictors that quantify criticality (or slack) rather than just produce a binary prediction.

We would like to thank the anonymous reviewers for their comments. This research was funded by NSF grant No. CCR-0105743 and SRC contract 2001-HJ-897.

References

[1] C. Alexander, D. Reese, and J. Harden. Near–critical path analysis of program activity graphs. In *Proceedings of the 2nd International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 308–317. IEEE Computer Society, Feb. 1994.

[2] B. Calder, G. Reinman, and D. M. Tullsen. Selective value prediction. In *26th Annual International Symposium on Computer Architecture*, pages 64–75, May 1999.

[3] J. Casmira and D. Grunwald. Dynamic instruction scheduling slack. In *2000 KoolChips workshop*, Dec. 2000.

[4] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *34th Annual International Symposium on Microarchitecture*, Dec. 2001.

[5] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *28th International Symposium on Computer Architecture*, July 2001.

[6] B. A. Fields, R. Bodík, and M. Hill. Slack: Maximizing performance under technological constraints. In *To appear in the Proceedings of the 29th International Symposium on Computer Architecture*, 2002.

[7] B. A. Fields, S. Rubin, and R. Bodík. Focusing processor priorities via critical-path prediction. In *Proceedings of the 28th International Symposium on Computer Architecture*, 2001.

[8] 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, Oct. 1999.

[9] F. Gabbay and A. Mendelson. Speculative execution based on value prediction. EE Department TR 1080, Technion - Israel Institue of Technology, Nov. 1996.

[10] A. Kaufmann and G. Desbazeille. *The Critical Path Method*. Gordon and Breach, 1969.

[11] M. Lipasti and J. Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1996.

[12] R. L. Martino. *Critical Path Networks*. MDI Publications, 1967.

[13] R. Rakvic, B. Black, D. Limaye, and J. P. Shen. Non-vital loads. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 165–174, Feb. 2002.

[14] J. Seng, E. Tune, and D. Tullsen. Reducing power with dynamic critical path information. In *Proceedings of the 34th International Symposium on Microarchitecture*, Dec. 2001.

[15] S. Srinivasan, R. Ju, A. Lebeck, and C. Wilkerson. Locality vs. criticality. In *Proceedings of the 28th International Symposium on Computer Architecture*, July 2001.

[16] S. T. Srinivasan and A. R. Lebeck. Load latency tolerance in dynamically scheduled processors. *Journal of Instruction Level Parallelism*, (1):1–24, 1999.

[17] D. M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[18] 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*, Feb. 2001.