In Proceedings of the 7th International Symposium on High Performance Computer Architecture, January 2001.

Dynamic Prediction of Critical Path Instructions

Eric Tune Dongning Liang Dean M. Tullsen Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
{etune,tullsen,calder}@cs.ucsd.edu

Abstract

Modern processors come close to executing as fast as true dependences allow. The particular dependences that constrain execution speed constitute the critical path of execution. To optimize the performance of the processor, we either have to reduce the critical path or execute it more efficiently. In both cases, it can be done more effectively if we know the actual instructions that constitute that path.

This paper describes Critical Path Prediction *for dynamically identifying instructions likely to be on the critical path, allowing various processor optimizations to take advantage of this information. We show several possible critical path prediction techniques, and apply critical path prediction to value prediction and clustered architecture scheduling. We show that critical path prediction has the potential to increase the effectiveness of these hardware optimizations by as much as 70%, without adding greatly to their cost.*

1 Introduction

Modern processors remove most artificial constraints on execution throughput. Out-of-order processors remove artificial dependences imposed by instruction ordering, register renaming removes false dependences, and aggressive branch prediction schemes greatly reduce serialization of instruction execution due to branches. Therefore, the bottleneck for many workloads on current processors is the true dependences in the code. Chains of dependent instructions constrain the overall throughput of the machine, often leaving aggressive processor technology highly underutilized. These chains of dependent instructions constitute the critical performance path, or *critical path* (CP), though the code.

The performance of the processor is thus determined by the speed at which it executes the instructions along this critical path. In our efforts to get the maximum performance from the processor, it is no longer reasonable to treat all instructions the same. If we can know which instructions are critical to performance, we can accelerate their execution, possibly at the expense of instructions not on the critical path.

Knowing which instructions are critical can allow the processor to improve performance by giving those instructions preference any time the processor needs to arbitrate between instructions. It can be used to give critical path instructions access priority to a variety of speculative hardware mechanisms. Architectures which have the potential to break or reduce the length of dependence chains (e.g., value prediction, instruction reuse) should target those dependence chains that are critical to performance.

In this paper we show that critical instructions can be effectively identified in hardware. We call this *critical path* (CP) prediction. This prediction is based on the behavior of previous invocations of the instruction in the pipeline. This prediction enables the processor to make better decisions about where to apply certain policies and optimizations. We examine several critical path predictors, and use these predictors to guide value prediction and instruction placement on a clustered architecture.

This paper is organized as follows. Section 2 introduces critical path prediction with a simple case study. Section 3 discusses related work. Section 4 describes our experimental methodology. Section 5 describes the general technique used by each of our critical path predictors, as well as several specific predictors. Section 6 evaluates the relative effectiveness of the predictors at identifying the critical path. Section 7 demonstrates two potential applications of the technique. Section 8 concludes.

2 Identifying Critical Path Instructions

This section uses a simple code example to demonstrate the importance of finding the critical path, and to give insight into how one might recognize those critical path instructions. Figure 1 shows the compiler-generated code for a simplified (for clarity) version of Livermore Loop 23, which has one loop carried dependence (besides the induction variables). This loop-carried dependence constitutes what we call the critical path through this code, and is shown in bold.

Code		SC IPC	IQ lat.	Oldest in IQ
ldt	f1, 8000(t3)	1.02	1	0
ldt	f10, 0(t1)	1.02	1	0
ldt	f11, 8(t3)	1.02	1	0
ldt	f12, 0(t4)	1.02	1	0
addq	t2,0x1, t2	1.02	1	0
cmplt	t2,a1, t7	1.02	2	0
lda	t1, 8(t1)	1.02	1	0
lda	t4, 8(t4)	1.02	1	0
lda	t5, 8(t5)	1.02	1	0
lda	t3, 8(t3)	1.02	1	0
mult	f1,f10, f1	1.02	3	0
ldt	**f10, -16(t3)**	**5.51**	**286**	**1**
mult	f11,f12, f11	1.02	4	0
ldt	f12, -8(t5)	1.02	1	0
addt	f1,f11, f1	1.02	8	0
mult	**f10,f12, f10**	**5.27**	**288**	**2**
ldt	f12, -8(t3)	1.02	1	0
addt	**f1,f10, f1**	**6.74**	**290**	**4**
subt	**f1,f12, f1**	**6.06**	**294**	**4**
mult	**f1,f0, f1**	**5.66**	**298**	**4**
addt	**f12,f1, f1**	**4.80**	**298**	**4**
stt	**f1, -8(t3)**	-	**302**	**4**
bne	t7, ...	-	1	0

Figure 1. Assembly code for a simplified version of Livermore Loop 23.

The example shows "SC IPC", or short-circuit IPC, next to each instruction. Short-circuit IPC is the throughput that this code achieves if the destination register of the corresponding instruction (and only that instruction) was correctly value-predicted for each iteration of the loop, eliminating its output dependences. These results demonstrate several principles:

- An optimization which breaks dependence chains is only effective if it does so along the critical path. Most instructions in this loop have absolutely no impact on performance (the IPC with no dependences removed is 1.02).

- Critical-path instructions, and their dependents, tend to get stalled in the instruction queue, and often become the oldest (bottom) instruction in the queue at some point. The columns labeled "IQ latency" and "Oldest in IQ" in Figure 1 show for each instruction, the average number of cycles spent in the instruction queue, and at the bottom of the instruction queue, respectively. These numbers correlate well with the critical path.

- Breaking the chain at any point along the critical path is effective.

- Instruction type is of limited value in identifying important instructions.

- Short-term dependence paths can be misleading. The longest path through a single iteration is different than the critical path in this case.

The last point is important, and motivates the approach taken in this paper. We choose not to attempt to explicitly track all dependence chains and identify the ones that matter. Rather, this example shows that the behavior of an instruction as it moves through the pipeline is a much more emphatic indication of where the critical path is, and is typically much easier to track. In fact, the most critical path is difficult to compute, in general, and depends on specifics of the processor. For example, if the instruction window of the processor is too small to hold an iteration of this example loop, the critical path through the loop changes significantly. While the dependences do not change to reflect this, the behavior of individual instructions will.

Several things make finding the critical path more difficult in the general case than in this example, particularly in the irregular applications we focus on; for example, irregular control flow, large instruction working sets, and more short-lived critical paths. Despite that, most of the principles identified here carry over to the general case.

We apply several heuristics to try and identify critical path instructions in this paper, most of which look for clues in the pipeline, such as those discussed here.

3 Related Work

Compiler-based critical path reduction optimizations have used dynamic analysis of the control flow of a program [3], followed by a static analysis of the data dependences through a single high-probability path or trace [20, 8, 24]. The prior work in compiler-based optimization concentrates on finding the most popular control trace/path through the program, using either edge or path profiling. Static profiles assume a certain popular control path based on the training inputs or other heuristics, and cannot account for changing program modalities, or varying processor implementations. The dynamic predictors in this paper can change their predictions over time. Also, the dynamic predictors can change execution behavior, yet require no ISA changes.

A prime application of critical path prediction is in guiding the use of techniques that can reduce the critical path by breaking dependence chains. Two such techniques are value prediction [18, 19, 10] and instruction reuse [25]. Calder, *et al.* [4] demonstrated that using the longest dependence chain in the current instruction window to guide which instructions should produce or consume predicted values can make value prediction more effective. That research proposed no hardware model for identifying or predicting these longest dependence chains, nor did it consider other mechanisms.

Bahar *et al.* [2], and Fisk and Bahar [9] identified loads which are not on the critical path in order to give prefer-

Benchmark	Input	Fast Forward
lisp	ref	1 000 000 000
compress	`bigtest.in`	1 000 000 000
go	`5stone21`	1 000 000 000
perl	`scrabbl`	1 000 000 000
ijpeg	ref	100 000 000
gcc	`1stmt.i`	0
burg	`rrh-mot`	0
delta-blue	long	0
mpegplay	`sukhoi.mpg`	100 000 000

Table 1. The benchmarks used in this study.

Parameter	Value
Fetch width	16 instructions per cycle
Branch predictor	Same as Alpha 21264
Branch Target Buffer	256 entry, 4-way associative
Active List Entries	1024
Functional Units	12 Integer (8 also load/store), 6 FP
Instruction Queues	128-entry Int, 128-entry FP
Registers	200 Int, 200 FP
Inst Cache	64KB, 2-way, 64-byte lines
Data Cache	64KB, 2-way, 64-byte lines
L2 Cache	4 MB, 2-way, 64-byte lines
Latency (to CPU)	L2 18 cycles, Memory 98 cycles (if no contention)
Instruction Latencies	Based on Alpha 21164

Table 2. Processor configuration.

ential cache placement for data accessed by critical loads. Zilles and Sohi [33] proposed identifying a few static instructions with the greatest impact on execution and pre-executing them.

Load hit-miss prediction [32] uses prediction structures, derived from branch predictors, to predict whether individual load instructions would hit or miss in cache, but the predictions are only used to schedule load instructions.

4 Methodology

Table 1 summarizes the benchmarks used in all our simulations. The first 6 benchmarks come from the SPEC 95 integer suite, and their inputs come from the reference set. These benchmarks are compiled with the DEC CC compiler at –O4. *Mpegplay* is an IBS benchmark [29]. *Burg* is a C++ parser generator. *Delta-blue* is a C++ constraint solution system. Both Burg and Delta-blue have significantly higher data cache miss rates than the other benchmarks. The benchmarks were fast-forwarded the number of instructions indicated in Table 1 before being simulated for 300 million instructions.

Execution is simulated on an out-of-order superscalar processor model which runs unaltered Alpha executables. The simulator is derived from [26]. This architectural simulator is enhanced to include a critical path predictor, and to take advantage of various critical path-aware optimizations. The simulator models all reasonable sources of latency, including caches, branch mispredictions, TLB misses, and various resource conflicts, including renaming registers, queue entries, etc.

The simulated processor configuration shown in Table 2 was used for the studies in Sections 6 and 7. The configuration models a future wide superscalar out-of-order machine, with an aggressive fetch unit, a large instruction window, and a large unified renaming unit. The L1 caches modeled are more modest, to compensate for the relatively small memory footprint of most of our benchmarks. The fetch unit can fetch up to 16 instructions per cycle from up to three basic blocks per cycle. This simulates the behavior of an effective trace cache [22].

The processor model used in our simulator has 9 stages. During the fetch stage, instructions and predictions which were requested in the previous cycle arrive. After decoding and register renaming, integer and floating-point instructions enter separate instruction queues. The instructions reside in the queues in-order. Every cycle, the oldest instructions which have their dependences satisfied are issued (out-of-order), until no more instructions are ready or no more functional units are available. They require one stage to read register values before they can begin execution. After execution, they go through one stage to write back registers. The instructions remain in the commit stage until they can be committed in order. This pipeline is similar in basic structure to the Alpha 21264, described in [5].

5 Critical Path Prediction Techniques

Critical path prediction consists of (1) marking instructions as having been on the critical path, and (2) predicting instructions to be on the critical path based on past markings. Both occur simultaneously in the processor. This section first discusses the Critical Path Prediction Buffer, which uses the past behavior of an instruction to predict its current behavior. We then discuss the actual criteria used to mark instructions as having been on the critical path.

5.1 Critical Path Prediction Buffer

Critical path prediction, like branch prediction and value prediction techniques, is based primarily on the previous history of an instruction. A PC-indexed table of saturating counters is updated according to an instruction's prior trips through the processor, and is queried when the instruction is next fetched.

For this research, all of our critical path prediction schemes follow the same process. In moving through the pipeline, an instruction meets a critical path *criterion* and that instruction is *marked*, indicating that this instruction may have been "critical". When and if the instruction commits, a saturating counter corresponding to that instruction is incremented if the instruction was marked, or decremented if it was not marked. When the instruction is next fetched, it

is *predicted* to be "critical" if the counter in a *Critical Path Buffer* (CPB) is above a threshold value. Otherwise it is predicted to be "not critical".

Throughout this paper we will use the following terms. An instruction is *predicted as critical* if its counter in the critical path buffer was above the threshold when the instruction was fetched. These instructions will have their *CP predicted bit* set, which will identify the instruction as a critical instruction for optimization purposes as it moves through the pipeline. *Marking criterion* or *criterion* means an event which causes instructions to be marked as future candidates for critical path prediction. This is implemented by having a *CP marked bit* set during execution for the instruction, and this bit is inspected when the instruction commits. A *marked* instruction is an instruction with this bit set. A committing instruction's marked bit is used to update the CPB, so that it can be predicted correctly in the future.

In understanding this new architectural technique, we want to separate the effectiveness of the technique from any aliasing effects that might occur in a small prediction table. Therefore, we assume a relatively large 64k-entry direct-mapped (indexed by PC, but untagged) table of 6-bit saturating counters for the CPB. The counters are incremented by 8 during commit when an instruction is identified as critical, and decremented by 1 when it is not. The predict threshold value is 8; when a counter *exceeds* the threshold, corresponding instructions are predicted as being critical path instructions. We investigated many other settings for increment, decrement and threshold, but found the above values to perform well for the critical path prediction heuristics we examined.

5.2 Critical Path Marking Techniques

In this paper, we propose five different criteria that might be used to mark each instruction as either on the critical path or not on the critical path. We evaluate each criterion individually; only a *single* CP criterion is applied during a particular simulation. Some are trivial to implement, others might be quite complex. Initially, we are more interested in what works than the complexity of the implementation.

The criteria are summarized in Table 3. What follows is a more detailed description of each criterion and the rationale behind it. This is actually a subset of the predictors we investigated, but includes those that were interesting either because of their performance or the intuitiveness of the approach.

The QOLD criterion is based on the observation that instructions on the critical dependence path will typically reach the bottom of the instruction queue before they issue. Any instruction which reaches the bottom of the queue becomes the oldest instruction. This instruction has dependences that exceed (in time) the dependences of all prior instructions in the instruction stream (for that queue, integer or floating point)

Criterion	Description
QOLD	"**OLD**est instruction in **Q**ueue" Each cycle, the oldest instruction in an instruction queue is marked, if it is not ready to issue.
QOLDDEP	"**DEP**endence with **OLD**est instruction in **Q**ueue" Every cycle, each instruction which produces a value consumed by the oldest instruction in the queue is marked if it is still active.
ALOLD	"**OLD**est in **A**ctive **L**ist" Each cycle, the oldest instruction in the active list (re-order buffer) is marked.
QCONS	"Most **CONS**umers in **Q**ueue" Each cycle, the instruction is marked whose result is used by the most instructions in the instruction queue.
FREED3	"**FREED** up at least **3** instructions in queue" If the completion of execution of an instruction makes at least three instructions in the instruction queue ready to execute, then the completing instruction is marked.

Table 3. The criteria used in this study to mark instructions as critical path, and a brief description of each.

Whereas QOLD marks the oldest instruction in an instruction queue, the QOLDDEP criterion marks the one or two instructions upon which it is dependent. In other words, if the instruction at the head of an instruction queue has source registers x and y, then we will try to mark the instructions which produce x and y. However, if x has already left the pipeline we do not mark it, since x's entry in the CPB would have already been updated when x committed. Therefore, QOLDDEP marks zero, one, or two instructions per cycle. This criterion attempts to mark the instructions that are currently causing instructions to back up in the instruction queues. This is one step earlier in the critical path dependence chain than the oldest instruction in the queue (QOLD).

The ALOLD criterion is based on the observation that the oldest active instruction in the machine is likely to be one that was stalled for some reason, either because of dependences or because it took a long time to execute. The active list has an entry for every instruction in the pipeline, waiting to commit in order. The oldest instruction in the active list is usually one that completed execution later than all prior instructions.

The QCONS criterion marks the one instruction, among those completing execution, which has the most direct consumers in the instruction queue. We define a consumer as an instruction that will read the value written by this instruction. In the case of a tie, the earliest instruction in the instruction stream is marked. The QCONS criterion is based on the observation that instructions that have a large dependence fan out are more likely to be on the critical path. Bahar, et. al. [9] tried measuring processor performance over very short time scales to allow the identification of non-critical loads, but found that counting the number of consumers of a load was a better metric. This corresponds to the QCONS criterion.

The FREED3 criterion is similar to the QCONS criterion, but it only counts consumers which become ready to execute immediately (they are *freed* by the executing instruction). This criterion is implemented as a threshold mechanism. It marks all instructions which free up 3 or more instructions in the instruction queue. The idea of scheduling instructions earlier which have a high fan-out has been applied to static instruction scheduling in compilers [12].

An instruction that stalls in the instruction queue or has a large execution latency is likely to accumulate more instructions in the queue waiting for its completion. Therefore, QCONS and FREED3 account for both the delay associated with an instruction's input dependence and the existence of critical output dependences. FREED3 and QCONS will obviously miss some instructions on the critical path that have only a single output dependence.

6 Evaluating Critical Path Predictions

Evaluating critical path prediction is more difficult than evaluating other prediction techniques. This difficulty stems from two significant differences between CP prediction and other predictors. First, in CP prediction it is more difficult to verify the accuracy of a prediction. Second, when CP predictions are used to direct optimizations, these optimizations will affect future CP predictions.

There are two steps in a branch predictor: prediction and verification. The true outcome of the branch is used to verify the prediction and to train the predictor. In critical path prediction, however, we can only verify whether the instruction again satisfied the criterion; we cannot verify whether or not the instruction was actually on the critical path. The predictor is only predicting that the criterion will be met again in the future. Therefore, for critical path prediction to work, we must meet two conditions. First, the predictor must accurately predict which instructions will meet the marking criterion. Second, the marking criterion must be a good heuristic method for identifying critical path instructions. In evaluating our techniques, we measure two different aspects of CP prediction. In section 6.1, we asses the *predictor accuracy*; how accurately does the predictor predict whether instructions will meet the marking criterion. In section 6.2 we measure the *criterion effectiveness*; how well do the predictions indicate which instructions are in fact critical.

The second difficulty may be referred to as the *feedback* problem. Namely, prior predictions affect future predictions. In bimodal branch prediction, the prediction used for the branch will not affect the update of the counter. In critical path prediction, an instruction that is predicted as critical will be optimized (e.g., value predicted, sent to a different cluster, etc.). After being optimized, it may no longer be on the critical path, and it may not be marked as critical. However, if it is subsequently not optimized, it may again appear on the critical path. This effect is discussed more in section 6.3.

Criterion	Percent Instr. Marked	Percent Instr. Predicted	Percent Non-CP Prediction Accuracy	Percent Positive Prediction Accuracy
QOLD	14	26	99	49
QOLDDEP	17	33	99	50
ALOLD	15	35	99	36
QCONS	6	16	99	36
FREED3	5	7	99	64

Table 4. The percent of executed instructions that each technique marks and causes to be predicted, as well as the accuracy with which each predictor predicted the same behavior used to mark instructions.

CP prediction is not an optimization, but an enabler for other optimizations. The absolute gains shown in this paper are strictly determined by the optimizations we choose to model and the constraints we place on them. It is only the change in the optimization's effectiveness that is interesting. For that reason, we define the *Effectiveness Ratio* (ER) as follows:

$$\mathrm{ER} = \frac{\mathrm{Speedup}_{\text{with CP prediction}} - 1}{\mathrm{Speedup}_{\text{without CP prediction}} - 1}$$

Therefore, if an optimization which provides a 20% speedup can achieve a 40% speedup when critical path prediction is incorporated, it has an effectiveness ratio of 2.0 – it has made the optimization twice as effective.

6.1 Measuring Prediction Accuracy

This section examines the degree of self-correlation (or repeatability) of the prediction criteria — that is, if event A is used to mark critical instructions and update the predictor, is the corresponding predictor actually a good predictor of event A? If not, it is unlikely to be a useful criterion.

To measure this self-predictability, the simulator was set only to mark and predict instructions; no actions were taken based on the predictions. What was measured is how often an instruction, which was predicted to be on the critical path, was again marked as a critical path instruction.

Table 4 shows the results for each CP algorithm, averaged over all benchmarks. The first column lists the names of the criteria tested, as described in section 5.2. The column labeled "Percent Instr. Marked" shows the percentage of dynamic instructions that had their CP marked bit set. The column labeled "Percent Instr. Predicted" shows how often any dynamic instruction had its CP predicted bit set. Remember that an instruction has its predicted bit set if its counter, in the PC-indexed Critical Path Buffer, is above 8. The column marked "Percent Non-CP Prediction Accuracy" measures what fraction of dynamic instructions that are predicted as "not on critical path" do *not* trigger the marking criterion again. The column marked "Percent Positive Prediction

Accuracy" measures what fraction of dynamic instructions that are predicted as being on the critical path have their CP marked flag set again the next time they are executed.

The results demonstrate that our predictors are intentionally liberal. One reason for this is to identify instructions only occasionally on the critical path. For example, on a load with a 20% miss rate that is only on the critical path when it misses, we might do best to always predict it on the critical path. This assumes that the cost of a wrong positive prediction is typically less than the cost of not predicting the instruction as being critical. Note that for the 65-93% of instructions predicted as not being on the critical path, the predictors are virtually always right.

6.2 Measuring Prediction Effectiveness

This section evaluates the effectiveness of our marking criteria in indicating which instructions are on the critical path. One approach would be to compute the critical path of a program by finding the longest chain of dependent instructions in a trace of the program, and to compare these instructions with those that are predicted by the CP predictor. There are several downfalls to this approach:

- The statically-determined critical path depends not just on dependences, but also on the idiosyncrasies of the processor, including queue sizes, active list size, number of renaming registers, and even on the input used when running the program.
- When the critical path information is used to optimize certain instructions, the optimizations can change the critical path, and the critical path predictor needs to adapt to the changes in the critical path caused by its previous predictions. The statically-determined critical path does not account for these changes.

To evaluate performance we will again use the approach from section 2, which focuses on the actual performance when the critical path prediction is used to change execution. In this section, we apply an ideal, generic optimization to compare several proposed predictors outside of the context of a specific optimization; the next section applies more realistic optimizations.

In this experiment, each cycle in which instructions are fetched, one instruction from the fetched block is chosen to execute with no output dependence stalls. That is, subsequent instructions that depend on this instruction will not have to wait for this instruction to execute. This emulates optimizations that break data dependence chains, such as value prediction and instruction reuse, but without presupposing exactly what optimization it is or which instructions it would work on. The choice of which instruction to select is based on the critical path prediction.

Figure 2 shows the speedup achieved on this test for the various dynamic predictors. The speedup is relative to the execution time with no optimization. We also provide the following measurements for comparison:

- FIRST: Always select the first instruction fetched this cycle.
- RANDOM: Pick an instruction randomly each cycle from the instructions fetched.
- STATIC: We precompute the critical path of the program by identifying the instructions which are on the longest chain of dependences in the program using profiling [27]. The profiler computes a dynamic critical path, accounting for cache and branch effects as well as a limited instruction window size. While a single, complete dynamic path is identified, the tool creates a static summary of each instruction's contribution to the dynamic critical path. The most critical static instructions, accounting for 98% of the dynamic path, are then statically identified as critical for the purposes of the STATIC predictor in these simulations. Each cycle, then, a statically marked instruction is chosen from the fetch block to be optimized, if possible.
- LONGEST: The instruction with the longest estimated execution latency is chosen. The latency is "estimated" because the latency of loads varies. The hierarchy of latencies we assume is based on Alpha 21264 latencies. We use a static estimate for load latency which places it lower than integer multiply and most floating point arithmetic operations, but above all other integer operations. For the integer-intensive applications shown here, then, LONGEST often amounts to "choose the first load." Exceptions are `mpegplay` and `ijpeg` which have a fair number of integer multiply and floating-point instructions. We also tested a different version of LONGEST which priorized loads over integer multiply and floating point instructions, but it did not perform as well.

We see that in almost all cases, the use of critical path prediction consistently results in greater speedup than the non-dynamic FIRST and RANDOM mechanisms. We found that on every benchmark, ALOLD, QOLD, QOLDDEP, and QCONS performed better than LONGEST. FREED3 was slightly worse on lisp and compress, but better then LONGEST on the other seven benchmarks. Additionally, on each benchmark, at least one of our dynamic predictors performed better than STATIC. This confirms that our dynamic predictors are adapting to changes in the critical path (chiefly caused by the optimizations themselves) in ways that the STATIC predictor cannot. Note that the benchmark and input files used to generate the static profile are identical to those used in the simulations. In a practical use of static profiling,

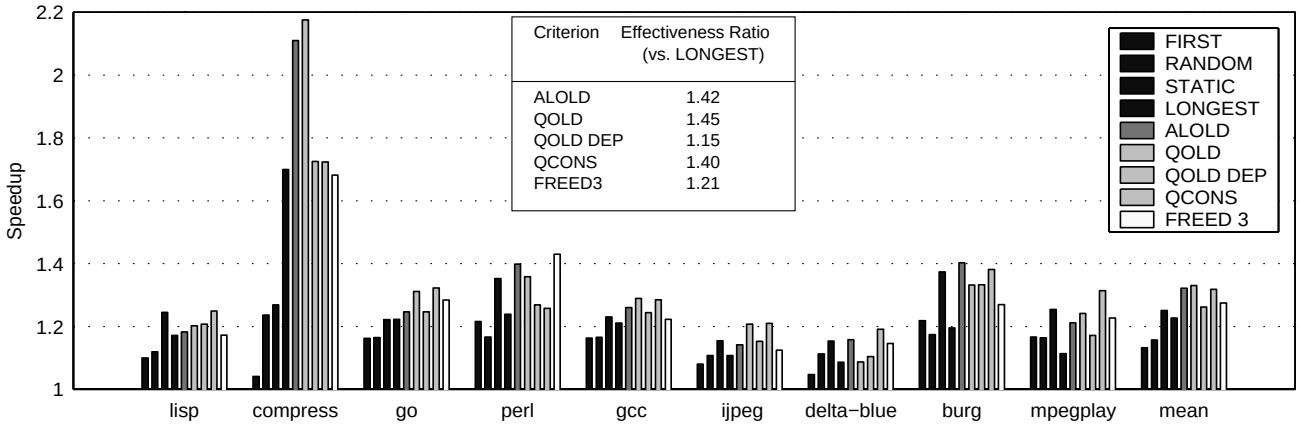

Figure 2. The performance resulting from breaking the dependences of critical path instructions.

differences between the inputs used for generating the static profile, and for actual execution would likely reduce the performance of the STATIC method. For the dynamic predictors, as well as STATIC, we assume we have the same information (estimated latency) available to use for the tie-breaker when multiple instructions, or no instructions, are predicted as being critical.

6.3 Counter Format and Prediction Persistence

In some cases, predicting an instruction as critical path (and applying some optimization) causes that instruction to no longer be on the critical path. However, this does not mean that we should no longer consider the instruction as critical. We'll refer to the predictor's natural inclination to start decrementing an instruction's CPB counter as *forgetting* a prediction. We can minimize the CPB's tendency to forget by incrementing the CPB counters by a large amount when an instruction is on the CP and decrementing by a small amount when not. In the previous experiments, we increment by eight and decrement by one, partially to avoid forgetting. Any instruction with a counter greater then eight has its predicted bit set. In the worst case, a CP instruction gets retried every eighth execution to confirm its criticality.

Not all of the marking criteria are affected in the same way. In particular, QCONS and FREED3 always forget because a successful optimization eliminates the dependences. On the other hand, when an instruction's result is, for example, value-predicted, that instruction must still execute to verify the prediction. Consequently, we would expect that ALOLD and QOLD would be less prone to forgetting. To verify this, we reran the dependence-breaking experiment of the previous section but with a a more forgetful counter, incrementing by two and decrementing by one. In these experiments, QOLD and ALOLD both performed better with the more forgetful counter, but the others (QOLD-DEP, QCONS, and FREED3) all performed better with the original increment-by-eight, confirming that they need the help of the confidence counters to force prediction persistence.

7 Using Critical Path Predictions

The previous section showed the potential for using critical path prediction by ideally removing a predicted instruction's dependences from execution. This section applies the predictions to more realistic optimizations. We first examine the benefit of using critical path information to guide value prediction. We attempt to get the best utilization out of a value predictor that is constrained in the number of predictions it can make. The second application uses critical path information to steer instruction placement in a clustered architecture.

7.1 Critical-Path Value Prediction

Critical-path prediction can assist value prediction in three ways. First, it allows the processor to make good choices when there are more predictable instructions in a fetch block than hardware resources to predict them. Second, it can be used to prevent costly misprediction penalties on instructions for which there is no benefit to prediction. Third, it can eliminate pollution in the value file by restricting which instructions are stored into it. Only the first benefit is examined in this paper.

Any reasonable value predictor will have limited prediction bandwidth. Gabbay and Mendelson [11] showed that prediction bandwidth is important for the performance of value prediction. They developed architectures to provide multiple value predictions per cycle, but at the cost of increasing the complexity and access time of the value prediction architecture. We take the opposite approach. We attempt to achieve the same performance out of a value prediction architecture by using critical path information with limited

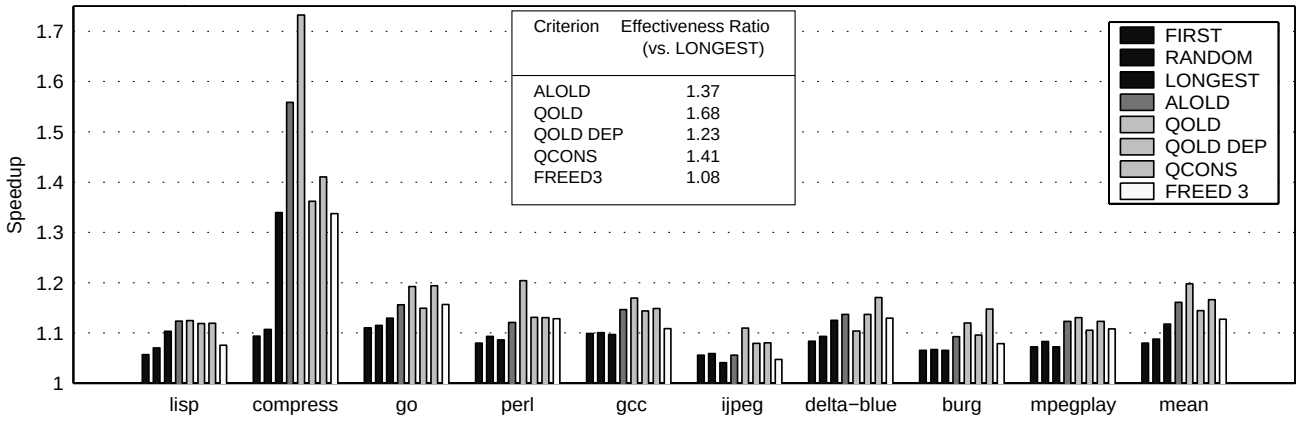

Figure 3. The performance of value prediction incorporating critical path prediction.

prediction bandwidth (in this case one value prediction per cycle).

Each cycle, value predictability information and possibly dynamic critical path predictions are supplied for each instruction fetched. Since these can each be a single bit, it is reasonable to assume the value confidence and CPB structures (which could be a single structure) have multiple read ports while the value file does not. Multiple value predictions also consume valuable register write ports.

If multiple instructions are marked as value predictable, one of several heuristics are used to select one for prediction. The heuristics are similar to those already shown. FIRST and LONGEST select the first or longest-latency instruction, and RANDOM selects a random instruction. The remaining bars show the performance when using a CPB with the specified CP prediction criterion.

The results (Figure 3) show that QCONS and QOLDDEP always provide more speedup than the selection schemes which do not use critical path predictions. QOLD delivers the best overall performance. It achieves an effectiveness ratio of 2.26 over the RANDOM selector (it has made value prediction 126% more effective) and an effectiveness ratio of 1.68 over LONGEST. The speedup observed for `compress` is much higher than with the other benchmarks, but the technique is effective in all cases.

Determination of value predictability for these experiments is idealized to account for the continued improvement of those techniques and confidence estimators. In particular, we assume perfect value prediction confidence. Therefore, if the instruction would be correctly predicted by either conventional last-value techniques [19], stride techniques [10, 13], or a context-based predictor [23, 31], we mark it as value predictable. We simulate aliased-free last-value and stride predictors. The context predictor is modeled after [31], with a 64K entry value history table, with four data values per entry.

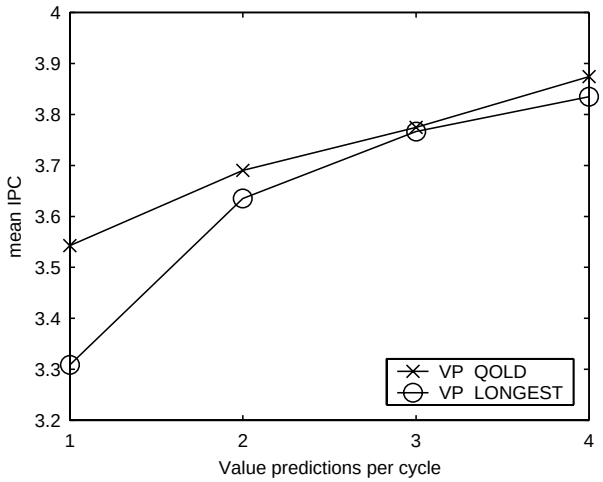

Figure 4. The performance of value-prediction of Critical-Path instructions for varying value-prediction bandwidth.

Figure 4 shows the results of using a value predictor that can provide 1, 2, 3 and 4 predictions per cycle. The same benchmarks and simulator were used for this experiment as for the last. We have selected the best performing criterion from the previous experiment with one value prediction per cycle. Namely, the top line shows the mean speedup over all benchmarks for QOLD. The lower line shows the speedup when the LONGEST selection scheme is used. The results show that the use of critical path information with 1 prediction per cycle bridges most of the gap between LONGEST with 1 and LONGEST with 2 predictions per cycle. With 2 predictions per cycle, the CPB still provides a noticeable increase over LONGEST. When more value predictions can be made per cycle, the two schemes start to converge, as the critical path arbitration becomes less necessary.

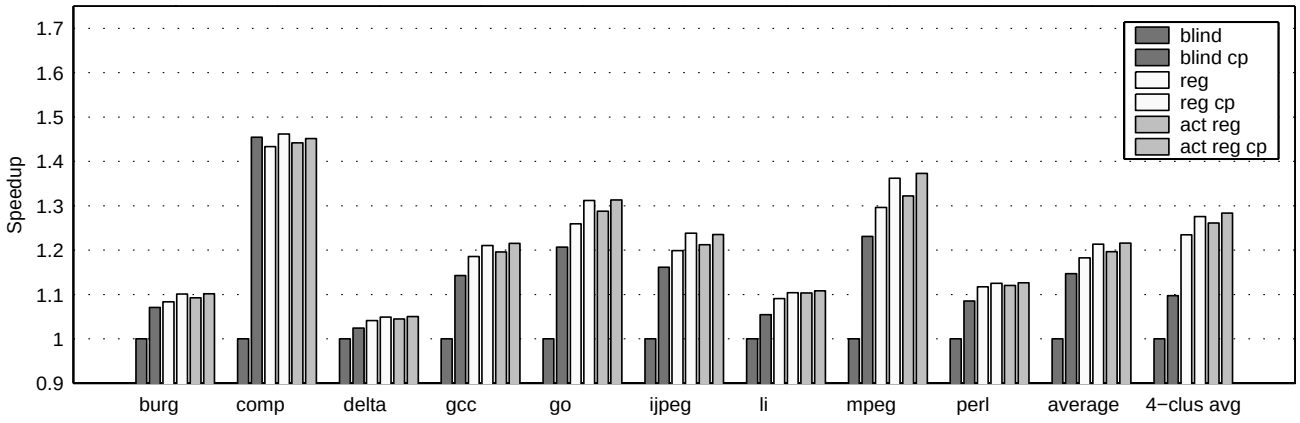

Figure 5. The performance of a critical path-aware clustered architecture.

7.2 Clustered Scheduling Architectures

Clustered architectures reduce the complexity and delay associated with key parts of the processor architecture core by separating functional units and associated structures into multiple groups. The clusters feature fast communication within a group, and slower between clusters. The Alpha 21264 [16] has two clusters of integer functional units, served by a duplicated register file, but a single instruction queue.

In this section, we simulate an architecture with two clusters of integer functional units, each served by a separate instruction queue. We assume bypassing of data between clusters takes 2 cycles longer than bypassing within a cluster. Instructions are assigned to a particular structure by hardware. This architecture is similar to that described in [15] and one of the machines described in [21]. A similar architecture is described by Farkas et. al., [6], but instruction scheduling is done statically. The M-machine [7] also features clusters, but their clusters are also not transparent to software.

Performance on a clustered architecture is optimized when the instructions at both ends of key dependences are assigned to the same cluster. Even better, we'd like to send an entire critical dependence chain through a single cluster. One simple way to achieve this is to always send predicted critical path instructions through the same cluster.

The processor model is the same as used for the previous experiments, except the integer queue is divided in half, each serving half of the integer/load-store functional units. We will examine three different heuristics for assigning instructions to clusters, with increasing degree of complexity, and each being modified to incorporate critical path prediction.

The first technique, **Blind** assignment, assigns instructions randomly, with its only priority being to balance the load in each queue. **Blind_cp** sends all CP-marked instructions to one cluster (if there is room); other instructions go to whichever cluster has more room. The blind algorithm suffers by not looking at register dependences, but has the advantage of allowing clustering to take place earlier in the pipeline, before such information is known, thus allowing more of the pipeline to benefit from decentralization.

The second technique, abbreviated **Reg**, takes register dependences into account. It attempts to send an instruction to whichever cluster the instructions providing the source operands were assigned. This is only violated when a queue is full or the queues are significantly out of balance. **Reg_cp** only uses critical path prediction to break ties when each operand comes from a different cluster.

The third technique, abbreviated **Act_reg**, is similar to *Reg*, but only considers the location of the producer of a source operand if that instruction has not yet completed execution (it is active). This is the mechanism closest to that assumed in [15] and [21], but is the most complex and assumes information not typically available to the early stages of the pipeline. **Act_reg_cp** again uses CP information to break ties when both operands are still waiting to execute.

The QOLD predictor was used for this application, in each case.

From Figure 5, we see that the critical path prediction data allows better assignment of instructions for the less complex assignment schemes, achieving an average 15% increase over the *Blind* scheme, but a smaller gain over *Reg*. That gain is enough to allow *Reg_cp* to overtake *Act_reg*, possibly allowing a less costly way to achieve the result, particularly if we are already using the critical path predictor for other uses. With *Act_reg* we find even fewer ties that need to be broken, but the small improvement shown even there demonstrates that we are still making the right decisions when given the opportunity.

The last set of bars show the same results for a 4-cluster architecture. In that case we see that the blind allocation

algorithm is more handicapped by the increase in clusters, but that the two register-based allocators are both more dependent on the critical path predictions to achieve their best performance.

7.3 Other Uses For Critical Path Prediction

Critical path knowledge can allow efficient use of various critical resources inside the processor. In the absence of resources to do an unlimited number of memory disambiguations or value predictions, we may still be able to get close to optimal performance with single-ported or dual-ported mechanisms if we guide the use of those resources through critical path prediction.

Multiple-path execution [30, 14, 17] follows both targets of conditional branches that have low prediction confidence. Better use of prediction resources could be obtained by not forking non-critical-path branches, or perhaps not forking branch directions that are not immediately on the critical path.

Critical path instructions can be given priority for issue when there are more data-ready instructions than there are functional units.

Multithreaded processors [1, 28] place higher pressure on issue bandwidth and other execution resources, and would therefore see higher benefit from a mechanism that used critical path prediction to manage resources, such as guiding instruction issue priority.

Various power optimizations would also be possible. Non-critical path instructions (e.g., loads) could be prevented from executing speculatively. The processor might choose to stall for some cycles during execution of a long-latency operation known to be on the critical path.

Data accessed by critical load instructions can be guarded against replacement [2, 9] in order to save energy and improve performance.

Several of these are the subject of ongoing or future research.

In this section we have discussed many potential applications of critical path computing, and have simulated two examples. These two applications validate our thesis that knowing the critical path can allow us to make useful tradeoffs between CP and non-CP instructions. These examples also validate the accuracy of our predictors, demonstrating that the predictors are indeed identifying important instructions, enabling significant performance enhancements.

8 Conclusions

This paper introduces the concept of critical path prediction, which seeks to identify those instructions that constrain the performance of the processor. In order to approach or even surpass the current true-dependence bottleneck, the processor needs to know exactly which instructions create that bottleneck. Critical path prediction information can be useful just about anywhere the processor must arbitrate between instructions, or where hardware structures are prone to contention or pollution.

We examine a variety of techniques to identify and predict CP instructions, including some which are simple to mark in the pipeline and provide excellent accuracy. We show that we can bias the processor in favor of critical path instructions and against other instructions and consistently achieve performance gains.

Critical path prediction is a technique which can be used to increase the effectiveness of other structures or optimizations. We demonstrate critical path prediction effectiveness on several hardware optimizations. First, we establish the potential for critical path prediction by applying it to an idealized optimization. Then we apply it to value prediction and clustered architecture instruction scheduling.

We demonstrate that the effectiveness of a value predictor can be more than doubled through the use of critical path prediction, relative to a value predictor that must select randomly among multiple instructions that are deemed to be predictable. It is 68% more effective than a value predictor that uses decoded instruction information to make the selection based on expected latency.

As processors increase their ability to exploit ILP in the instruction stream, application performance becomes more tied to the execution of the critical dependence path. Optimizations that accelerate critical path execution will have an increasingly large advantage.

Acknowledgments

We would like to thank the anonymous reviewers for their useful comments. This work was funded in part by NSF CAREER grants No. MIP-9701708 and No. CCR-9733278, NSF grant No. CCR-980869, and a grant from Compaq Computer Corporation.

References

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The tera computer system. In *International Conference on Supercomputing*, pages 1–6, June 1990.

[2] R. Bahar, G. Albera, and S. Manne. Power and performance tradeoffs using various caching strategies. In *International Symposium on Low Power Electronic Design*, Aug. 1998.

[3] T. Ball and J. Larus. Efficient path profiling. In *29th International Symposium on Microarchitecture*, Dec. 1996.

[4] B. Calder, G. Reinman, and D. Tullsen. Selective value prediction. In *26th Annual International Symposium on Computer Architecture*, May 1999.

[5] Compaq Computer Corp., Shrewsbury, MA. *Alpha 21264 Microprocessor Hardware Reference Manual*, Feb. 2000.

[6] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The multicluster architecture: reducing cycle time through partitioning. In *30th International Symposium on Microarchitecture*, Dec. 1997.

[7] M. Fillo, S. Keckler, W. Dally, N. Carter, A. Chang, Y. Gurevich, and W. Lee. The M-Machine multicomputer. In *28th Annual International Symposium on Microarchitecture*, Nov. 1995.

[8] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

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

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

[11] F. Gabbay and A. Mendelson. The effect of instruction fetch bandwidth on value prediction. In *25th Annual International Symposium on Computer Architecture*, 1998.

[12] P. Gibbons and S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, 1986.

[13] J. Gonzalez and A. Gonzalez. The potential of data value speculation to boost ilp. In *12th International Conference on Supercomputing*, 1998.

[14] T. Heil and J. Smith. Selective dual path execution. Technical Report http://www.engr.wisc.edu/-ece/faculty/smith_james.html, University of Wisconsin, Madison, Nov. 1996.

[15] G. Kemp and M. Franklin. PEWs: A decentralized dynamic scheduling algorithm for ILP processing. In *International Conference on Parallel Processing*, 1996.

[16] R. Kessler, E. McLellan, and D. Webb. The alpha 21264 microprocessor architecture. In *International Conference on Computer Design*, Dec. 1998.

[17] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the PolyPath architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[18] M. Lipasti and J. Shen. Exceeding the dataflow limit via value prediction. In *29th International Symposium on Microarchitecture*, Dec. 1996.

[19] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *17th International Conference on Architectural Support for Programming Languages and operating Systems*, Oct. 1996.

[20] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, R. Nix, J. ODonnell, and J. Ruttenberg. The multiflow trace scheduling compiler. *Journal of Supercomputing*, 7(1-2):51–142, May 1993.

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

[22] E. Rotenberg, S. Bennett, and J. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *29th Annual International Symposium on Microarchitecture*, December 1996.

[23] Y. Sazeides and J. Smith. The predictability of data values. In *30th International Symposium on Microarchitecture*, 1997.

[24] M. Schlansker and V. Kathail. Critical path reduction for scalar programs. In *28th International Symposium on Microarchitecture*, Nov. 1995.

[25] A. Sodani and G. Sohi. Dynamic instruction reuse. In *24th Annual International Symposium on Computer Architecture*, June 1997.

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

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

[28] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[29] R. Uhlig, D. Nagle, T. Mudge, S. Sechrest, and J. Emer. Instruction fetching: Coping with code bloat. In *22nd Annual International Symposium on Computer Architecture*, 1995.

[30] S. Wallace, B. Calder, and D. Tullsen. Threaded multiple path execution. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[31] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *30th Annual International Symposium on Microarchitecture*, Dec. 1997.

[32] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation techniques for improving load related instruction scheduling. In D. DeGroot, editor, *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*, May 1999.

[33] C. Zilles and G. Sohi. Understanding the backward slices of performance degrading instructions. In *27th Annual International Symposium on Computer Architecture*, June 2000.