Computing Along the Critical Path

Dean M. Tullsen Brad Calder

Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114

UCSD Technical Report
October 1998

Abstract

Modern processors come close to executing as fast as true dependencies allow. The particular dependencies 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 path.

This paper describes a Critical Path Profiler *for efficiently finding the critical dependence path through the complete execution of a program. It is intended to be used for profile-based compiler and processor optimizations. Unlike other critical-path based tools which analyze dependences along a single most-likely path (trace), this one analyzes dependences along every taken path through the code. We present data on the SPEC benchmark suite gathered through the profiler and experiment with potential uses for the profiler as a static critical-path predictor.*

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 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 on the critical path, we can accelerate their execution, possibly at the expense of instructions not on the critical path.

Value prediction, branch prediction, static instruction scheduling, dynamic instruction scheduling, as well as fetch and issue on a multithreaded CPU can all benefit from knowledge of which instructions are on the critical path.

This paper focuses on static techniques to determine the dynamic critical path of execution through a program. Other papers, which attempted to compute the inherent ILP in a group of programs [4, 11, 22, 26], did so by measuring (using our terminology) the *length* of the critical path. This paper is more concerned about measuring the actual components of the critical path — which instructions they are, and what are the characteristics of those instructions?

In addition, we have created a critical path profiler which is fast enough to be used for feedback into the compilation process. We will describe the profiler, provide information on the instructions found to be on the critical path, and describe some possible applications of critical-path data. While this paper focuses on static critical path prediction, it is also intended to demonstrate opportunities for using critical-path knowledge, paving the way for both static and dynamic critical-path prediction techniques.

This paper is organized as follows. Section 2 discusses related studies. Section 3 describes the profiler, including the algorithms used, it's accuracy, and it's execution efficiency. Section 4 presents results that measure the length of the critical path under different architectural assumptions. Section 5 examines more specifically the components of the critical path, and looks at specific characteristics of instructions on the critical path. We suggest some potential uses of critical path data in Section 6. Section 7 concludes.

2 Related Work

This study has several things in common with studies that attempt to measure the inherent instruction-level parallelism (ILP) limits in various programs. They profile or simulate code, following the dependence paths, and measure the amount of parallelism given various architectural constraints. Among those have been Smith, et. al., [20], Butler, et. al., [4], Wall [26], Theobald, et. al., [22], and Lam and Wilson [11].

The difference in this paper is that we are not searching just for the length of the critical path (another way of thinking about the ILP they measured), but the composition of the critical path. We want to know exactly what instructions constitute the critical path, and as much as we can about the characteristics of those instructions. This information is useful in both compiler and processor optimizations, as shown in [2, 5, 25].

Traditionally, critical path-reduction optimizations have been done through a 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 [14, 7, 19]. The prior work concentrates on finding and optimizing the most popular *control* trace/path through the program, found using either edge or path profiling. In contrast, our approach concentrates on finding and optimizing the critical *data paths* through the complete execution of a program taking into account variable-latency instructions such as loads and branches, and architectural delays (e.g., instruction window size).

3 Profiling the Critical Path

If we are to compute the critical path efficiently enough to be used in a profiling context, it would take too long to collect the entire execution trace and post-process it, searching for the longest chain of instructions. This section

Figure 1: A snapshot of the dynamic dependence graph that might be captured by the critical-path profiler with a 4-instruction window.

describes a profiling algorithm for computing the critical path of execution using constant storage and execution time proportional to the number of instructions profiled.

The critical path profiler uses a very simple processor model keeping track of (1) an instruction window, (2) memory hierarchy, (3) branch prediction, and (4) instruction execution latencies. Instruction data latencies are from the Compaq Alpha 21164. Loads use a cache model to determine their latency. For this paper, all results assume 64KB I and D caches and a 4 MB L2 cache, all 2-way set-associative. Both register and memory data dependencies are identified. However, not all dependencies are data dependencies. All instructions following a branch misprediction are dependent on the branch (misprediction penalty 7 cycles; gshare branch prediction; 2K-entry PHT; 1K, 4-way BTB).

As the program executes, the profiler keeps track of instructions in the current instruction window (IW). Instructions enter the IW in order, and exit the IW in order. The IW defines the window of parallelism within which instructions can issue out of order — analogous to the instruction queue or reservation stations. It also tracks instructions recently removed from the instruction window (which are then placed in what we call the trace window, TW). The profiler maintains dependences between instructions in the IW or TW as graph edges. These dependence edges are removed if the edge is no longer relevant to current computation (there is no path back to an instruction within the instruction window). In addition, when the *use* of a dependence edge E is older than the *def* end of any other dependence edge (i.e., there are no other overlapping edges), then edge E has met the *retirement criteria*. At that point, edge E is retired as a permanent portion of the critical path. Specifically, the instruction that is the source of the edge is counted on the critical path for a time equal to the length of the dependence.

This process is described in the context of Figure 1, which shows dependences between nine instructions around a four-instruction (small for the purpose of illustration) IW. The initial instruction window is shown on the left between the horizontal lines. We only allow 4 instructions to be in the current window at a time. When *i5*'s dependences are satisfied, it issues, allowing *i9* to enter the window, and the instruction window moves from the region shown on the left to the region shown on the right. Edges *i6-i9* and *i8-i9* (*i9*'s dependences) are added. Now there is no longer a path from *i2* to the current IW, so edges *i2-i4* and *i4-i5* are removed. Edge *i1-i3* is

now the oldest edge, with no overlapping edges, so it has met the retirement criteria. It is recorded as part of the critical path and also removed.

Instructions *i1*, *i2*, *i4*, and *i5* are all removed from the trace window because they have no edges in the graph (and are not in the IW). *i3* still has a relevant edge, and remains in the graph.

In practice, more than data dependences constrain program execution; architecture features can also cause delays in a program. A stalled instruction can delay other instructions from entering the instruction window if the instruction window is full. This type of delay can be on the critical path through the program, and need to be modeled. To represent the delays (e.g., instruction cache miss and full instruction window) that prevent instructions from entering the IW, we add *window-edge* dependencies between the instructions that want to enter the IW and those that are preventing them from entering.

Occasionally, our trace window reaches its maximum size without an edge meeting the criteria for retirement as a critical path edge. In this case, tie-breaking measures are taken that remove edges that overlap with the oldest edge. Our results indicate that such measures are not taken often.

Calculating the critical paths by keeping track of the edges that meet the retirement criteria does not guarantee a complete coverage of "execution time." For example, there are periods when no critical path edges assert themselves long enough to be counted (e.g., when several competing edges are removed at once). Therefore, the summed length of the edges of the critical path is not the same as expected execution time (which is not the point of the profiler, but part of the analysis of this paper). Fortunately, our mechanism for moving instructions in and out of the instruction window maintains a strict definition of time, including the current cycle. It is this value that is used to calculate total CP length and IPC in Section 4.

The profiler produces various outputs (including several added just for this study), including lists of the top static instructions that constitute 80%, 90%, 95%, or 98% of the accounted-for critical path (CP). We will refer to these lists as, for example, the 80%-CP. Even the 98%-CP is an effective filter of the total critical path, typically being well under 1% of the size of the 100%-CP list, which includes all CP instructions (even those only on the CP once). One of the profiler's outputs is a human-readable record of the critical path over a disassembly of the program. An example of a small code fragment from *applu* is shown in Figure 2.

The results after the colon, from left to right, are the number of times the instruction was executed, the number of times it was on the critical path, the number of cycles it contributed to the critical path, the percentage of the total CP instruction count, and its percentage of the total CP length. No data is printed for instructions that are never on the CP.

Profiled executables are from the SPEC95 benchmark suite and compiled with the Digital Unix C and FORTRAN compilers for the Alpha with full optimization. All profiles for this paper either run the program to completion or stop at 10 billion instructions profiled.

3.1 Profiler Accuracy

For this tool to be useful for profile-driven program optimization, it must have two properties: profile accuracy, even across different inputs, and reasonable profile execution time. This sections examines the former, show-

20018018	ldt $f10, 0($4)	:11491062	5744117	11488234	1.149%	0.238%
2001801c	ldt $f12, 0($17)	:11491062	5744119	11488238	1.149%	0.238%
20018020	ldt $f15, 0($5)	:11491062	2826	5652	0.001%	0.000%
20018024	subq $7,1, $7	:				
20018028	lda $4, 8($4)	:				
2001802c	lda $5, 40($5)	:				
20018030	mult $15,$10, $10	:11491062	5746943	22987772	1.150%	0.476%
20018034	subt $12,$10, $10	:11491062	11491062	45964248	2.299%	0.952%
20018038	stt $f10, 0($17)	:11491062	11491062	57455310	2.299%	1.190%
2001803c	bgt $7, 0x20018018	:				
20018040	ldt $f19, 0($1)	:9575885	964	1928	0.000%	0.000%
20018048	xor $17,$31, $17	:9575885	707	707	0.000%	0.000%

Figure 2: An example of one of the outputs of the critical-path profiler.

ing that profile data is valid across data sets, and that the profiled data itself is a valid indicator of instruction importance.

To determine the variability of the critical path across data sets, we profiled each of the benchmarks with a training data set that was different than the reference data set being used to this point. Then we calculated the portion of the reference data set critical path that was covered by the 98%-CP, 95%-CP, and 90%-CP instruction lists from the training profile. For example, the 95%-CP list of the training run would ideally be close to 95%. Table 1 shows that the correlation between runs is extremely high with only two exceptions. In 15 of 17 applications, the critical path is highly independent of the data inputs. However, we show in Section 6.1 that even the other two applications still can benefit significantly (e.g., see Figure 9) from using the training-set generated profiles.

We verify the critical path profiler by comparing the critical path found during profiling to instructions that cause stalls during a detailed cycle-by-cycle instruction-level simulation of an Alpha processor [23]. If our profile is accurate, we expect it to identify a high percentage of the stall-producing instructions in a program running on a machine with similar instruction window size and cache parameters. While not all stall-producing instructions are on the critical path, just about all critical path instructions should cause some kind of stall; therefore, we expect a high correlation between the critical path instructions and stalling instructions.

This allows us to test the validity of the profile in a simulation environment that makes different and certainly more complex assumptions about the architecture than the profiler. While the profiler's assumptions were designed to match up with the simulated processor, there are key differences in the nature and size of the instruction scheduling window, layout and alignment of both heap and stack variables, instruction fetch limitations, TLB and other latencies, number of functional units, as well as memory hierarchy bandwidth limitations. The simulated processors fetches up to eight instructions per cycle, issues up to six integer and three floating point, and models the processor pipeline and all sources of latency in detail. It has a 32-entry integer instruction queue and 32-entry floating point queue. We begin simulation a billion instructions into the program and simulate another billion (the profiles cover the first 10 billion instructions).

Benchmark	ref data	train data	Coverage of ref CP		
			98% CP	95% CP	90% CP
applu	ref	train	96%	92	85
apsi	ref	train	92%	90	87
comp	ref	train	95%	89	76
fpppp	ref	train	98%	95	89
gcc	1cp-decl.i	amptjp.i	98%	95	91
go	5stone21	2stone9	95%	90	82
hydro2d	ref	train	98%	96	91
ijpeg	ref	train	98%	95	90
li	ref	train	95%	91	86
m88ksim	ref	train	95%	91	85
mgrid	ref	train	62%	61	60
perl	scrabbl	jumble	47%	40	33
su2cor	train	test	98%	94	88
swim	ref	train	98%	95	92
tomcatv	train	test	90%	90	88
turb3d	ref	train	98%	95	91
wave	ref	train	95%	91	85

Table 1: Profiler accuracy across different inputs. This table shows the coverage of the reference data set critical path using the 98% critical path, 95% critical path, and 90% critical path instructions generated by the profiler-training data set.

In this experiment, we measure the percentage of time that the oldest instruction in either of the two (fp, integer) instruction queues is stalled, and is dependent on an instruction on the critical path. While other instructions could be stalled, the oldest is most likely to be a throughput-constraining stall. The stall contribution shown (the second number in each field of Table 2) is the percentage of cycles in which the oldest instruction in either the floating point queue or the integer queue was stalled waiting for a critical-path instruction identified by the profiler, as a percentage of all cycles in which one of them was stalled waiting for any instruction. The first number is the percentage of all dynamic instructions that are marked as critical path instructions.

For these experiments, we used the 100%-CP, 98%-CP and 95%-CP lists, as well as three approximations to the 98%-CP list. All of these lists are generated with the reference data set, and verified with simulation using the same data set. We observe from Table 2 that the 100%-CP list is generally too large to be useful, as nearly every important static instruction is on the critical path at least once, and thus makes that list. The 98%-CP list has much more desirable characteristics—low percentage of dynamic instructions, but still a high percentage of stalling instructions. The critical-path approximation algorithms in some cases identified more stalling instructions than the full profile. This was typically true when the approximate methods produced larger lists (after the 98% filter was applied) than the original.

Investigating the lower correlation for turb3d and su2cor revealed that most of the stalled instructions in the integer or floating point queues that were not marked as CP-dependent by our profiler were actually *store* or *branch* instructions. Store and branch instructions typically end a dependence chain, thus the instructions causing

benchmark	% of all instructions / % of cycles a CP inst causes stall				
			98%-CP		
	100%-CP	95%-CP	full	5% samp.	20% samp.
applu	85.5 / 100	18.3 / 82.4	14.5 / 76.9	22.8 / 84.8	20.8 / 84.8
apsi	84.3 / 100	10.7 / 65.9	8.8 / 60.5	14.0 / 75.9	13.3 / 72.2
compress	78.9 / 100	24.8 / 78.7	24.4 / 78.4	24.9 / 78.7	24.8 / 78.7
fpppp	85.5 / 100	8.4 / 65.5	7.2 / 58.2	11.7 / 73.5	11.6 / 73.5
gcc	71.7 / 99.8	35.8 / 82.9	40.8 / 88.0	40.6 / 87.8	41.1 / 88.0
go	78.5 / 99.9	49.8 / 89.4	44.3 / 83.8	48.9 / 88.3	49.3 / 88.8
hydro2d	85.3 / 100	23.9 / 68.5	17.2 / 62.7	29.0 / 78.6	27.6 / 77.3
ijpeg	88.7 / 99.6	26.1 / 66.7	20.8 / 60.2	27.8 / 69.2	27.6 / 68.7
li	63.3 / 100	29.4 / 83.9	37.2 / 90.7	39.0 / 92.2	38.0 / 92.4
mgrid	96.5 / 100	27.7 / 83.1	23.4 / 78.4	30.5 / 89.6	29.8 / 85.2
perl	69.3 / 99.9	30.1 / 78.1	24.0 / 70.0	31.6 / 79.0	31.6 / 79.0
su2cor	73.8 / 100	18.1 / 63.2	17.9 / 62.9	19.3 / 66.7	18.4 / 64.8
swim	88.3 / 100	28.9 / 81.0	23.0 / 66.4	33.9 / 87.4	31.8 / 86.3
tomcatv	88.7 / 100	28.2 / 85.9	23.0 / 77.4	41.4 / 93.2	36.5 / 95.2
turb3d	81.5 / 99.9	11.5 / 42.7	8.3 / 35.2	14.5 / 55.4	14.1 / 54.6
wave	80.2 / 100	23.1 / 56.8	19.3 / 54.2	27.4 / 64.7	27.6 / 71.1

Table 2: Correlation of the profiled critical path with instructions that cause the oldest instruction in the machine to stall

those stalls were *correctly* being identified as not on the critical path using our profiler. A secondary effect was that memory data was not aligned in the same way.

These results show that our critical-path profiler is effectively identifying instructions which are important to the performance of the executing program, which ultimately is *more* important than completely accurately identifying the critical path. More importantly, we prove the profile by demonstrating its validity by showing in Section 6.1 that optimizations benefit from using the critical path profiles.

3.2 Profiler Efficiency

The tool is built on the Compaq ATOM [21] tools, and profiling 10 billion instructions can take a significant amount of time. Therefore, we examined the viability of sampling as a technique to reduce profile runtime. Sampling simply neglects profiling a fraction of executed instructions. Because we are tracking paths of dependencies, we need to sample instructions in consecutive sequences rather than one at a time. For these measurements (Table 3), we profile 5000 instructions at a time, then turn off profiling for a time (e.g., for 5% sampling, we then skip 95000). For this data, we do keep the cache up-to-date for loads and stores that are otherwise not profiled; however, further performance could be gained by not doing the cache updates, with some degradation in accuracy. In this table, the 98%-CP instructions from the sampling run are used to compute the coverage of the reference run.

The reference data set is used for both the sampling and reference runs compared in the first two data columns. The accuracy lost in sampling is negligible.

Benchmark	Coverage of full CP		Runtime (seconds, on training input)			
	20% sampling	5% sampling	full	20%	5%	# insts
applu	98	98	245	81	50	269M
apsi	99	99	1404	480	273	1.43B
comp	98	98	36	13	8	45M
fpppp	99	99	225	81	52	234M
gcc	98	98	652	236	149	768M
go	98	98	422	185	136	493M
hydro2d	98	98	3774	1242	761	4.22BB
ijpeg	98	98	1471	512	320	1.81B
li	98	97	150	56	37	189M
m88ksim	98	96	90	34	24	118M
mgrid	98	98	8629	3126	2010	8.92B
perl	98	98	2304	962	705	2.68B
su2cor	99	98	10308	3373	1913	10B
swim	98	98	366	140	94	407M
tomcatv	98	98	1085	410	283	1.28B
turb3d	99	99	6951	2525	1623	8.75B
wave	99	98	1768	702	462	1.88B

Table 3: The coverage of the full-profile critical path for the 98%-critical-path instructions using 20% sampling and 5% sampling. Also given is the adjusted profile time for the training inputs.

We also provide in Table 3 the execution time of the profiler on a 500 MHz Alpha 21164. In the initial coding of the profiler, we have made no effort to optimize the code for efficiency. A significant part of the overhead is imposed by the ATOM tools. For example, when between sampling intervals, we could not turn off the instrumentation calls – they just return without doing any work. To at least partially account for this overhead, the times shown for profiling the training runs are adjusted by subtracting out the time for a skeleton ATOM executable that just counts instructions. Several opportunities exist for more efficient profiling, including instrumentation calls per basic block rather than per instruction, more efficient data structures for statistics recording, etc. A very significant contributor to runtime is the detailed models of the caches and branch predictors. Simpler, more crude models could reduce runtime at a cost of accuracy.

4 The Length of the Critical Path

The profiler's main function is to determine the critical path to enable compiler and hardware optimizations. However, it also computes the total length of the critical path, making it a useful tool for analysis of the applications and the assumed architecture. In this section, we calculate the ILP available for the SPEC95 applications under varying architectural assumptions using the critical path profiler. All results are given in IPC, instructions per cycle, for easier comparison with previous ILP studies. For our purposes, *IPC is defined as the ratio of the number of instructions executed divided by the total critical path length.*

This section examines two factors that impact the instruction-level parallelism available under the limited assumptions of the profiler: the size of the instruction window and the fetch bandwidth.

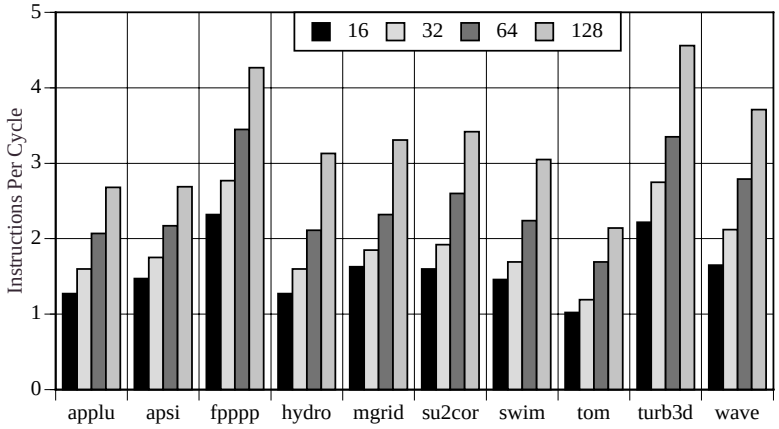

Figure 3: The effect of the instruction window size on the length of the critical path (expressed as IPC) for SPEC FP programs.

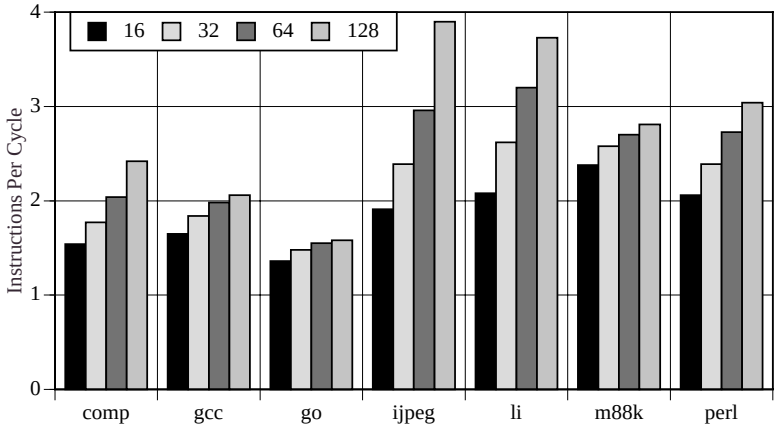

Figure 4: The effect of the instruction window size on the length of the critical path for SPEC Integer programs.

	CP segment length in insts				CP segment length in cycles			
	Instruction window size				Instruction Window Size			
	16	32	64	128	16	32	64	128
applu	5.1	6.3	10.4	18.0	28.9	38.8	75.6	140.7
apsi	7.0	9.4	14.1	23.9	36.5	58.1	86.0	164.4
fpppp	6.8	17.5	23.8	33.3	23.3	65.7	90.6	132.9
hydro	4.7	5.9	6.9	9.3	31.6	47.8	61.4	75.8
mgrid	22.8	45.9	7.4	7.3	149.0	325.8	71.1	60.9
su2cor	10.3	14.8	16.9	22.1	35.9	63.1	70.3	87.6
swim	4.9	4.3	7.4	3.6	31.6	42.0	79.1	64.1
tom	7.1	9.6	15.9	51.5	45.5	75.3	125.9	489.8
turb3d	5.1	8.5	17.1	12.9	17.8	30.7	61.2	40.7
wave	8.2	12.8	20.9	27.5	31.2	48.6	73.5	101.8
fp avg	8.2	13.5	14.1	21.0	43.1	79.59	79.5	135.9
comp	15.1	24.8	41.0	96.9	34.9	66.3	106.9	270.6
gcc	13.7	27.7	83.8	328.4	34.8	73.9	228.1	885.9
go	20.7	54.2	199.6	1023.1	50.3	136.3	509.9	2614.4
ijpeg	8.9	14.9	34.3	140.2	21.9	40.9	110.5	387.5
li	6.6	11.0	20.6	54.6	16.5	29.3	57.2	148.9
m88k	14.1	44.3	112.4	191.9	27.2	87.4	223.9	384.6
perl	9.0	18.4	48.6	191.9	20.6	43.6	119.7	481.1
int avg	12.6	27.9	77.2	289.6	29.5	68.3	193.7	739.0

Table 4: The average length, in instructions and cycles, of the continuous critical path segments, as the size of the instruction window is varied.

4.1 Window Size

The size of the instruction window determines the number of instructions that can be considered for simultaneous or re-ordered execution. In modern dynamically-scheduled processors, the instruction queue or reservation stations determine the number of instructions that can be scheduled together.

Figures 3 and 4 show the effect on critical path length of changing the instruction window size from 16 instructions to 128 instructions for the SPEC floating point and integer programs, respectively. These figures indicate that parallelism is *highly* dependent on the size of the instruction scheduling window. There is no indication in these results that parallelism levels off as the number of instructions allowed in the scheduling window increases. Increasing the effective size of the scheduling window will become a key element in the success of future processors' quest for increased ILP.

4.2 CP Segment Lengths

Another statistic that varies greatly with the window size is the average length of a continuous segment of the critical path. Given infinite parallelism and infinite look-ahead, a program would have only a single critical path (ignoring paths of equal length), and this single path would be the bounding limit on performance (assuming no critical-path changing transformations). However, given limited execution resources and a limited window of parallelism, the critical path is not a single path, but a collection of critical path segments. Over time, one segment

may dominate, then be replaced by a new path segment which is independent of the first. The longer the path segments the processor is finding, in most cases the closer we are to executing the program's single critical path and reaching our performance bound. Table 4 shows how the instruction window size determines the average length (in instructions and in cycles) of the segments found by our profiler. We compute this by calculating how often the *def* of the currently retiring CP edge is the same as the *use* of the last one (this may include non-data dependences like branch mispredictions and window-edge dependences).

In most cases the segment lengths increase significantly (both in number of instructions and cycles) as the window size increases. A large instruction scheduling window allows the processor not only to find more parallelism, but also to follow the true critical path more closely. Small instruction windows inject more discontinuities in the critical path by creating more non-data dependencies that interfere with the processor's ability to follow the true critical path.

4.3 Fetch Bandwidth

A limited fetch bandwidth constrains how effectively we can fill the instruction window to expose ILP. Other results in this paper assume unlimited fetch bandwidth, always keeping the instruction window full in the absence of instruction cache misses.

Figures 5 and 6 show the results of increasing the fetch bandwidth from 2 instructions per cycle to 32. These results all use an instruction window of 64 instructions. For this study, we only limit the raw fetch bandwidth, ignoring cache line boundaries and taken branches; therefore, these results represent an upper bound even for a trace cache [17] or other instruction fetch optimizations [6].

The object of the critical-path profiler, however, is not the length of the critical path, but rather identifying the instructions on the critical path. An analysis of the instructions that make up the critical path follows in the next section.

5 The Composition of the Critical Path

This section examines the actual instructions that compose the critical path, breaking them up by opcode, instruction type, memory behavior, branch behavior, value predictability and other characteristics.

5.1 Instructions on the critical path

In Figures 7 and 8, we identify the actual instructions on the critical path by type. These are shown for several different instruction window sizes, which alters the amount of parallelism available. In these figures, *long integer* refers to integer multiply, and *long fp* refers to fp divide. Long-latency instructions (load miss, integer multiply, all floating point) clearly represent a large portion of the critical path.

Both the makeup of the critical path and how it changes with the IW size varies widely in these applications; however, a few trends emerge. Factors that do not improve with increased window size (branch and jump mispredictions, primarily) become increasingly important as the total critical path length decreases with a large

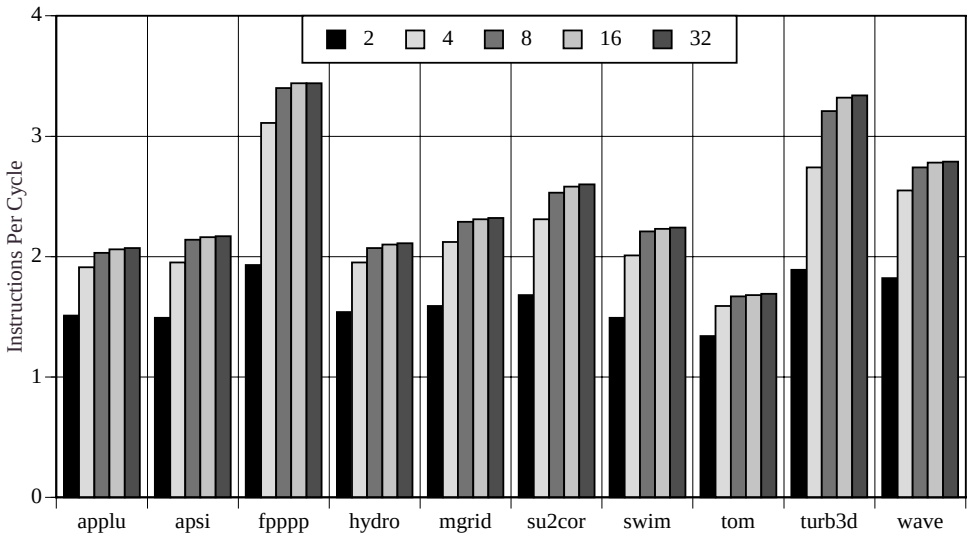

Figure 5: The effect of the fetch bandwidth on the length of the critical path (expressed as IPC) for the SPEC FP programs.

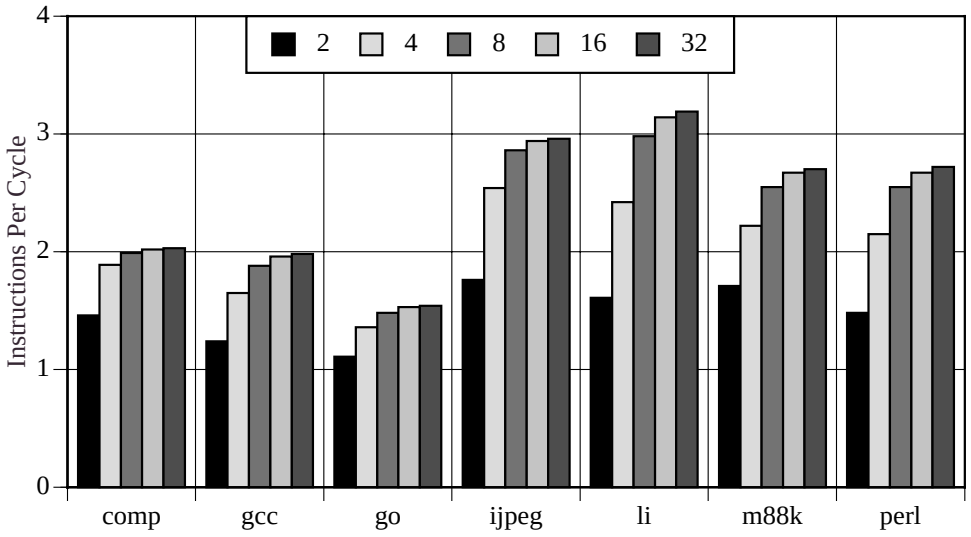

Figure 6: The effect of the fetch bandwidth on the length of the critical path for SPEC Integer programs.

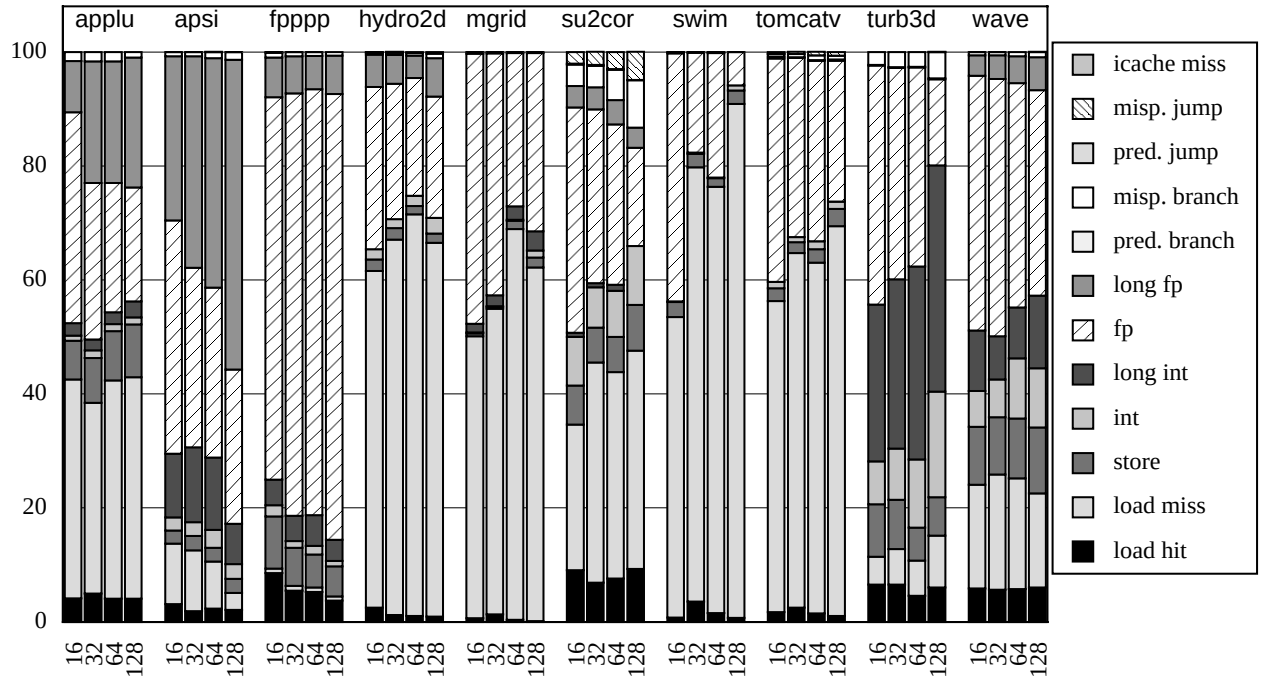

Figure 7: The effect of the instruction window size on the makeup of the critical path for SPEC FP programs.

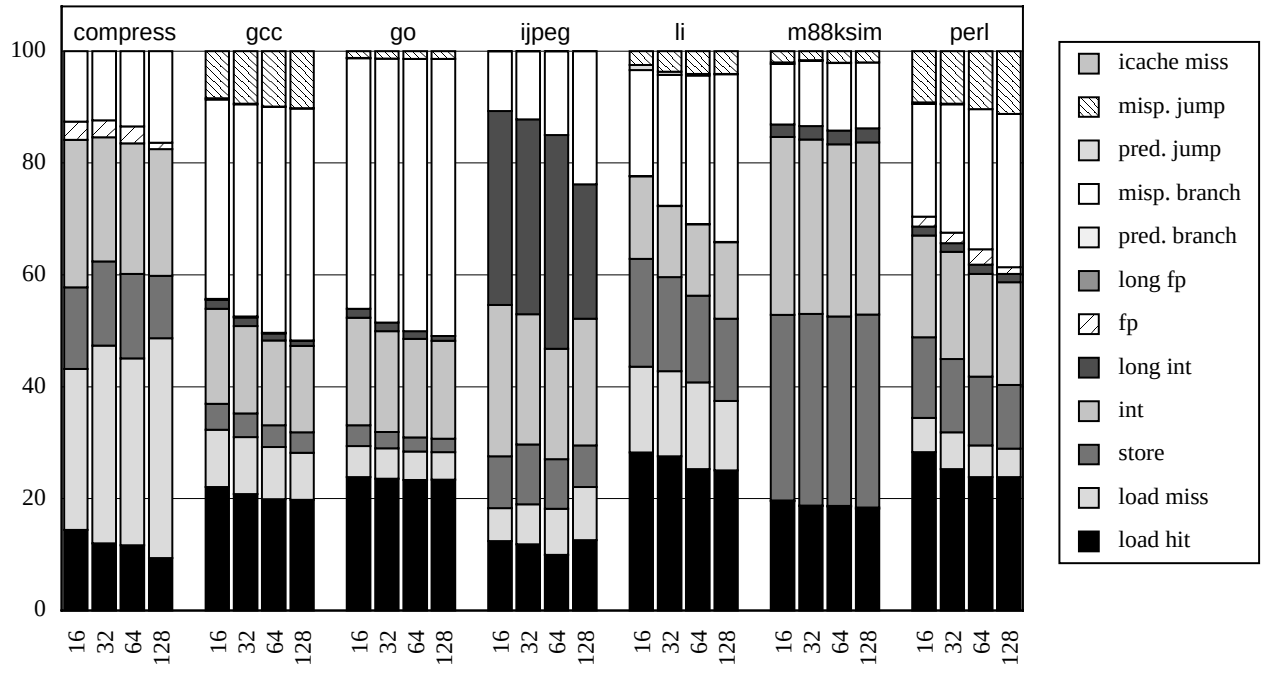

Figure 8: The effect of the instruction window size on the makeup of the critical path for SPEC Integer programs.

	FP SPEC			Integer SPEC		
	% IC	%CP	expected CP cycles	%IC	%CP	expected CP cycles
load hit	25	3.4	.05	22.9	18.9	0.3
load miss	1	38.5	13.6	0.4	11	13.7
store	10.4	4.6	.1	11.4	13.2	0.4
int	21.4	4	.05	49.8	19.7	0.2
long int	0.2	6.6	14.9	0.2	6.4	18.1
fp	37.9	33.1	0.3	0.4	0.8	1.5
long fp	0.4	8.1	5.9	0	0	-
pred branch	3.3	0	0	9	0	0
mispred branch	0.1	1.4	4.1	1.8	25.9	6.3
pred jump	0.4	0	0	3.9	0.1	0
mispred jump	0	0.4	5.6	0.2	4	6.0

Table 5: The expected CP contribution of an individual instruction to the critical path, in cycles.

IW. Also, when a single factor is a dominant part of the critical path (e.g., more than 50%), it consistently becomes more dominant as more parallelism is exposed, as other factors become more effectively hidden behind the dominant one. The FP applications seem to fall into three categories, memory-constrained (hydro2d, mgrid, swim, tomcatv), floating-point limited (apsi, fpppp), and balanced (applu, su2cor, turb3d, and wave). The integer applications are all relatively balanced compared to the FP.

This figure presents aggregate data for all instructions of a given type. However, if the processor or compiler is making a decision about a particular instruction, the expected importance of the instruction itself is more important than the aggregate contribution of that type of instruction. Table 5 gives the *expected CP contribution* of an instruction of each class. The table shows (1) the percent of executed instructions of that type on the critical path, (2) the percent of CP cycles the instruction type takes up, and (3) the average CP cycles for each instruction type. The average CP cycles for a given instruction type is calculated by taking the number of cycles those instructions account for on the CP, and dividing this by the number of executed instructions of that type. For example, while load misses and regular floating point instructions contribute almost equally to the FP SPEC critical path, a single load miss carries over 40 times the CP weight of a single floating point instruction.

The expected CP-importance of an instruction varies widely by type. In situations where our profiling techniques are impractical, instruction type can be used to make a good first-order prediction of an instruction's critical-path contribution, using this data. In [5], we exploit this in certain experiments by value-predicting only long-latency instructions like loads, which we have shown here to have high CP probability.

We can potentially make even finer distinctions between the expected importance of instructions of the same type based on opcode. We examine whether different instructions, with the same type and latency, vary enough in average CP contribution that we may be able to treat them differently.

Table 6 shows the expected critical-path impact of various branch instructions. The "expected CP cycles" shows the average number of cycles each instruction will contribute to the critical path. For example, each executed `load quad` for Int SPEC programs will contribute on average .67 cycles to the CP. From this we

	FP SPEC exp. CP	Int SPEC exp. CP
beq, bne	.10	1.05
blt, bgt, ble, bge	.19	1.32
blbc, blbs	.12	1.32
fp branches	.01	-
load long	.38	.50
load quad	.20	.67
load float	.46	.30
integer add		.24
all other short int		.15

Table 6: Branch, load, and short integer importance by opcode.

see a real variance between FORTRAN and C. For the FORTRAN code, a branch on inequality instruction has almost twice the CP importance of branch on equality (beq, bne); for the C code, the per-instruction importance is about equal for the two. The branch-on-bit does not show particularly different behavior than the other integer branches. For the floating point branches, the importance to the critical path is very low. Section 6 discusses further branch and critical path interactions.

Also in Table 6 are the relative importance of loads separated by data type. Again, the results vary significantly by language. For the FORTRAN programs a quadword load is only half as important as another type of load; however, for C programs the quadword load is significantly more important than longword loads.

We also broke down the integer instructions (for the integer applications) more finely and found that add instructions carry a large part of the load, as their importance ratio is 60% higher on average than other integer instructions (besides multiply).

5.2 Value Prediction and the Critical Path

Value prediction [13, 8] is a hardware optimization specifically aimed at exceeding the performance bounds of the critical path. Therefore, the value-predictability of the critical path is of particular interest, since this technique is only beneficial to the extent that it does attack the critical path.

Table 7 shows the value predictability of instructions on the critical path given last-value prediction [13, 8, 12], stride-based value prediction [8], and two-level context-based value prediction [18, 28]. Because we are most interested in the inherent predictability of individual instructions on the critical path, we assume unlimited table sizes for last-value prediction (LVP) and stride-based prediction (Stride). For the two-level context predictor (Context), we assume a scheme based on [28] with four values stored per instruction (unlimited total storage) and a very large 64K-entry shared pattern table. Results for Stride and Context include only those instructions that did not exhibit LVP predictability.

Table 7 shows the contribution to the total CP length of all instructions that exhibit more than 80% value predictability during the execution of the program. This table indicates that there is likely to be very little correlation between overall predictability and performance. We see several programs with high predictability,

Benchmark	CP influence of insts with value predictability					
	Last Value		Stride		Context	
	%IC	%CP	%IC	%CP	%IC	%CP
applu	10.3	4.2	3.8	0.7	9.6	10.6
apsi	11.9	7.7	11.1	8.5	4.9	5.3
fpppp	17.5	2.6	0	0	3.9	7.1
hydro2d	56.6	78.8	17.1	1.5	1.6	0.4
mgrid	2.8	24.1	16.3	2.1	0	0
su2cor	7.4	4.9	1.7	2.1	2.6	4.2
swim	0.9	0	15.8	2.4	0	0
tomcatv	7.0	10.1	8.3	0.7	0.9	0.6
turb3d	23.3	5.7	7.1	6.6	13.8	15.0
wave	27.4	24.7	8.0	5.3	1.1	0
fp avg	16.5	16.3	8.9	3.0	3.8	3.6
comp	2.5	0.2	0.4	0	1.0	0
gcc	29.9	10.9	4.1	1.7	5.0	2.3
go	24.2	6.4	.6	.6	3.8	2.2
ijpeg	10.4	3.1	10.9	2.6	0.7	0.5
li	23.0	9.3	3.2	10.8	10.8	10.2
m88ksim	4.5	4.5	0.3	0	0.7	0.1
perl	28.6	14.3	0.7	0.4	13.5	14.2
int avg	17.6	7.0	2.9	2.3	5.1	4.2

Table 7: The (aggregate) contribution of value-predictable instructions to the critical path.

but little impact on the critical path, and a few instances of programs with relatively low predictability, yet high CP importance of those predictable instructions. We see that in general the predictability of instructions is lower on the critical path than off of it (particularly if you ignore hydro2d's effect); however, there is still enough value-predictability on the critical path for value prediction to be very beneficial. Stride prediction and context prediction provide lower (but not insignificant) returns for the additional hardware, particularly when the impact on the critical path is considered.

The next section shows the effect of using critical-path information to guide value prediction and other hardware optimizations.

6 Using Critical Path Information

6.1 Applying Critical Path Knowledge

The previous sections focused on what we discovered when we applied critical-path profiling to the SPEC benchmarks. This section examines what we can do with that information once we have it.

There are a variety of ways in which we can exploit critical-path information. This section provides a few examples of hardware optimizations that might exploit CP information. Profile-generated critical path information could be used to drive compiler transformations such as height reduction and if-conversion [19], but that is not the focus of this paper. Critical-path information can be communicated to the hardware either through dynamic

critical path analysis (the subject of future work) or through the instruction set architecture (e.g., through new opcodes or marker instructions).

Value Prediction The previous section showed that it is unwise to assume that all value-predictable instructions are on the critical path. A decision to predict an instruction should be made not just on its predictability, but also on its CP importance. Predicting instructions not on the critical path can put the instruction on the critical path if it is mispredicted, or put more pressure on critical resources (e.g., predict bandwidth, reservation stations, value history tables) even if correctly predicted. In both cases, performance can be lost if the instruction is not on the critical path. These issues and others are explored much further in [5], which shows that even very simple critical-path information can be used to improve value prediction performance by being more selective. In particular, that study showed that (1) being selective about which instructions consumed a prediction in some cases eliminated recovery action for mispredicted values, and that (2) being selective about which instructions could produce a predicted value significantly eliminated conflicts in the value history table, thus increasing prediction accuracy.

This section explores yet another application of critical-path information to aid value prediction: the effective use of limited prediction bandwidth. Figure 9 shows the effect of several algorithms to choose which instruction to predict when *only one instruction per cycle can be value predicted*. Specifically, we make a choice from each block of instructions fetched in the same cycle. The algorithms measured are *first* (choose the oldest/first instruction fetched), *random* (choose one randomly), *latency* (choose the instruction with the longest latency–all loads are assumed to be 3 cycles), *CP-first* (choose the first CP instruction, if there is one, otherwise just the first), *CP-latency* (choose the CP instruction with the longest latency), and *CP-length* (choose the instruction which statically has the largest total contribution to the critical path). The last scheme requires more information to be recorded per instruction than whether it is on the critical path, but the extra information proves useful. These measurements, and all others in this section, use the processor simulation environment described in Section 3.1. For these experiments, however, we assumed a more aggressive processor design–a 16-wide processor with the ability to fetch up to three basic blocks per cycle (e.g., using a trace cache [17] or similar structure). All results in this section use the training profiles to guide execution of the reference runs.

These results show that the critical-path information, despite being generated under very different assumptions, is quite useful in directing the use of limited resources. In some cases, favoring long-latency instructions provided a useful approximation for the critical path, but the actual critical-path information provided even better performance. Knowing the actual contribution to the critical path allowed even better decisions (with one significant exception, compress).

An alternate type of value prediction, register-based value prediction, is described in [25]. This is a technique that allows the compiler to have a strong influence on the value predictability of individual instructions through changes to the register allocation. Register value prediction can take advantage of the critical path profiles prepared for this study to determine which opportunities to create reuse should be exploited.

Instruction Issue Priority In Table 8, we show the effect of incorporating critical path information into the hardware instruction scheduling mechanism. These simulations are run on the same processor simulator used

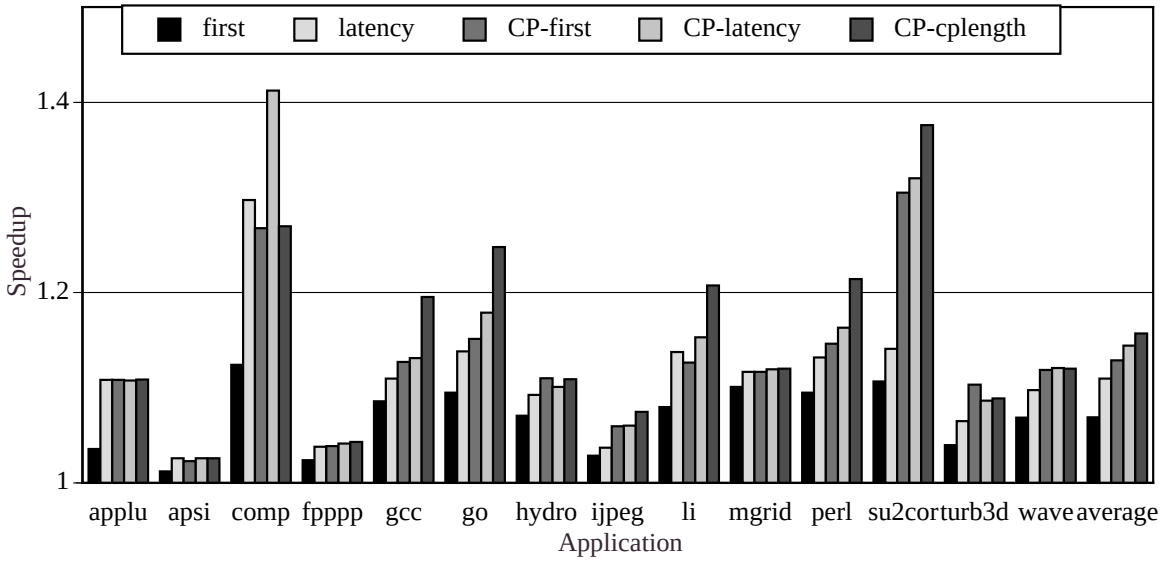

Figure 9: The performance of limited prediction bandwidth using alternative models to select one instruction per fetch block to be predicted. These results make the ideal assumption of perfect confidence.

Benchmark	Speedup
applu	1.008
apsi	1.000
comp	1.024
fpppp	1.001
gcc	1.038
go	1.005
hydro2d	1.004
ijpeg	1.035
li	1.046
m88ksim	1.004
mgrid	1.035
su2cor	1.025
swim	1.034
tomcatv	1.011
turb3d	1.012
wave	1.005

Table 8: Speedups obtained on a 5-issue processor using critical-path information rather than instruction order as the primary factor in instruction-queue scheduling.

previously, but modeling a smaller configuration. The processor is capable of issuing 3 integer (2 load/store) and 2 floating point instructions per cycle. The fetch bandwidth is still eight instructions per cycle.

In these simulations the processor gives first priority to instructions that are known (statically) to be on our 98%-CP critical path. In several cases large speedups are achieved with this simple hardware change.

Branch Prediction Perhaps the most intriguing branch-related results from our profiles is not the number of mispredicted branches on the critical path, but rather the number that are not. In fact, on average for the SPEC benchmarks we are running, 29% of mispredicted branches are not on the critical path. That means the branch was mispredicted, the following instructions squashed, and the (important) correct-path instructions are still brought into the machine before their operands were ready. Even more significant, we found that many individual branches have the property that they are often (relative to current branch prediction standards) mispredicted, yet rarely on the critical path. It may make sense, then, to handle those branches differently than other branches. We could statically predict them, to reduce dynamic predictor contention, or it would be beneficial to use Pipeline Gating [15] for those mispredicted branches not on the critical path, in order to save power. This would result in the processor deciding not to predict these branches and instead stall the fetch engine until the branch is resolved.

Other applications 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 per cycle, or even branch predictions per cycle, we may still be able to get close to optimal performance with single-ported mechanisms if we guide the use of those resources through critical-path prediction.

Multiple-path execution [27, 9, 10] 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.

Multithreaded processors [1, 24] place higher pressure on issue bandwidth and branch prediction resources, and thus would see higher benefit from both the instruction issue priority and branch prediction optimizations discussed previously in this section.

A clustered processor dedicates specific functional units to each of multiple instruction-scheduling windows [16]. Instructions assigned to different clusters experience longer bypass delays than those assigned to the same one. In that case it is critical that both the *def* and *use* end of each critical path edge go through the same cluster. This can be done simply by sending all critical path instructions through the same cluster. Depending on whether cluster assignment is done dynamically or statically (e.g., assignment is based on destination register value), either dynamic or static critical-path prediction can be used.

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

In this section we have discussed many potential applications of critical-path computing, and have simulated three examples. These simple applications validate our thesis that knowing the critical path can allow us to make

useful tradeoffs between CP and non-CP instructions. Both static and dynamic critical-path prediction techniques could be used to exploit these types of opportunities.

These examples also validate the accuracy of our profiler, as we are selecting the right critical-path instructions, even though the simulator has a much more complex model of hardware constraints, the memory subsystem, TLBs, branch prediction mechanisms, etc., as well as running on different input.

7 Conclusions

This paper introduces the concept of critical-path computing, which exploits knowledge of the critical dependences in a program's execution to optimize its performance. To support critical-path computing, we have developed a tool to create a static profile of the dynamic critical path through the program. Unlike traditional critical path tools used in compiler analysis which calculate a single most-likely path (trace) through basic blocks, this tool computes the critical-path dependences for every path through the code that is taken during execution.

Several results are shown from the output of the profiler. We examine the total length of the critical path under different architectural assumptions. We also look at the actual instructions that constitute the critical path, grouped in several ways. We examine characteristics of these workloads that could be useful even in the absence of specific profiles; for example, an integer multiply instruction's expected contribution to the critical path is about 100 times that of other integer ALU operations.

We present several applications for critical path computing. We simulate three, value-prediction under prediction-bandwidth limitations, instruction issue priority, selective application of static branch prediction, with a much more sophisticated instruction-level simulator. We show that we can bias the processor in favor of critical path instructions and against all other instructions and consistently get performance gains. We also show that the simplicity of the profiler still allows a complex processor to make correct decisions regarding instruction importance.

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 a large advantage. One tool that will help make that happen is a static predictor of critical-path importance such as the one described here.

Acknowledgments

This work was funded in part by a grant and equipment support 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.I. Bahar, G. Albera, and S. Manne. Power and performance tradeoffs using various caching strategies. In *International Symposium on Low Power Electronic Design*, August 1998.

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

[4] M. Butler, T.Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction steam parallelism is greater than two. In *18th Annual International Symposium on Computer Architecture*, pages 276–286, May 1991.

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

[6] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *22nd Annual International Symposium on Computer Architecture*, pages 333–344, June 1995.

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

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

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

[10] A. Klauser, A. Paithankar, and D. Grunwald. Selective eager execution on the polypath architecture. In *25th Annual International Symposium on Computer Architecture*, page To appear, June 1998.

[11] M.S. Lam and R.P. Wilson. Limits of control flow on parallelism. In *19th Annual International Symposium on Computer Architecture*, pages 46–57, May 1992.

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

[13] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. Value locality and load value prediction. In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

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

[15] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: Speculation control for energy reduction. In *25th Annual International Symposium on Computer Architecture*, June 1998.

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

[17] E. Rotenberg, S. Bennett, and J.E. Smith. Trace cache: a low latency approach to high bandwidth instruction fetching. In *29th International Symposium on Microarchitecture*, pages 24–34. IEEE, December 1996.

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

[19] M. Schlansker and V. Kathail. Critical path reduction for scalar programs. In *28th International Symposium on Microarchitecture*, pages 57–68, Ann Arbor, MI, November 1995. IEEE.

[20] M.D. Smith, M. Johnson, and M.A. Horowitz. Limits on multiple instruction issue. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 290–302, 1989.

[21] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 196–205. ACM, 1994.

[22] K.B. Theobald, G.R. Gao, and L.J. Hendren. On the limits of program parallelism and its smoothability. In *25th Annual International Symposium on Microarchitecture*, pages 10–19, December 1992.

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

[24] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[25] D.M. Tullsen and J.S. Seng. Storageless value prediction using prior register values. In *26th Annual International Symposium on Computer Architecture*, pages 270–279, May 1999.

[26] D.W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 176–188, April 1991.

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

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