Appears in the 2006 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2006)

Critical Path Analysis of the TRIPS Architecture

Ramadass Nagarajan Xia Chen Robert G. McDonald Doug Burger
Stephen W. Keckler

Computer Architecture and Technology Laboratory
Department of Computer Sciences
The University of Texas at Austin
cart@cs.utexas.edu - www.cs.utexas.edu/users/cart

Abstract

Fast, accurate, and effective performance analysis is essential for the design of modern processor architectures and improving application performance. Recent trends toward highly concurrent processors make this goal increasingly difficult. Conventional techniques, based on simulators and performance monitors, are ill-equipped to analyze how a plethora of concurrent events interact and how they affect performance. Prior research has shown the utility of critical path analysis in solving this problem [5, 18]. This analysis abstracts the execution of a program with a dependence graph. With simple manipulations on the graph, designers can gain insights into the bottlenecks of a design.

This paper extends critical path analysis to understand the performance of a next-generation, high-ILP architecture. The TRIPS architecture introduces new features not present in conventional superscalar architectures. We show how dependence constraints introduced by these features, specifically the execution model and operand communication links, can be modeled with a dependence graph. We describe a new algorithm that tracks critical path information at a fine-grained level and yet can deliver an order of magnitude (30x) improvement in performance over previously proposed techniques [5, 18]. Finally, we provide a breakdown of the critical path for a select set of benchmarks and show an example where we use this information to improve the performance of a heavily-hand-optimized program by as much as 11%.

1 Introduction

Modern processors exploit fine-grained concurrency from potentially hundreds of instructions in flight. Each instruction passes through a myriad of hardware resources resulting in numerous microarchitectural events during the course of its lifetime—cache misses, branch mispredictions, re-order buffer stalls, port contentions, and data dependence hazards. The active instruction window in a processor could thus feature thousands of events, some of which occur on concurrent paths, while others are dependent on each other. Naturally, some of them affect the overall execution time more than others. Understanding the interactions among these events and identifying the bottlenecks is important not only for designing balanced machines with the right hardware mix and capabilities, but also for providing accurate bottleneck-causing program profiles to an optimizing compiler.

Conventional simulation-based techniques and hardware performance-monitoring techniques are ill-equipped for a detailed performance analysis. While the coarse-grained view that they provide—the number of cache misses, branch predictions, or even execution profiles—can point designers in the right direction, they are insufficiently powerful to identify the finer-grained interactions and bottlenecks among a large set of concurrent and inter-dependent events [6]. Simulation-based techniques also quickly implode under the wake of numerous design parameters and the resulting combinatorial expansion of the design space.

Prior research has shown the utility of critical path analysis in solving the above problem [5, 18]. This analysis abstracts the execution of a program with a directed acyclic graph constructed using a simulator or a run-time profiler. Nodes in the graph represent microarchitectural events that occur during the lifetime of the program, while edges represent the dependence constraints among the events. These constraints include both data dependences among the instructions and machine constraints specific to the architecture. Different insights can be gained by analyzing the dependence graph. For example, one can identify if long execution times are a result of poor instruction-level parallelism (ILP) in a program. If the critical path—defined as the longest path in the graph—consists of a large fraction of data dependence edges in the program, then it is because of low available ILP. A different composition of the critical path may indicate other constraints. One can obtain the relative critical path contribution of each type

of dependence constraint and identify the potential bottlenecks among them. Researchers have shown a variety of such dependence-graph based analyses and their applications to understand and improve the performance of a processor [4, 5, 6, 13, 17, 18].

This paper extends the simulation-based critical path framework developed for conventional out-of-order processors and uses it to analyze the performance of a high-ILP processor architecture. The TRIPS architecture can support wider-issue microarchitectures than has historically been feasible [2]. It uses an execution model that treats large blocks of instructions as atomic units for fetch, execution, and commit. The ISA facilitates a distributed microarchitecture in which numerous computation tiles communicate using a routed network. We identify the new microarchitectural events and dependence constraints introduced by these features and show how they can be represented in the dependence graph.

The complexity of critical path analysis depends on the instruction window size of the processor. The TRIPS prototype processor with its 16-wide issue, 1024-entry instruction window, and distributed microarchitecture increases the complexity considerably. The complexity of the analysis also depends on the number of different types of dependence constraints and the granularities at which they are tracked by the graph. In the simplest form, only aggregate critical path contributions of a constraint may need to be tracked, for example the number of data cache miss cycles that appear on the critical path. However, for a better bottleneck analysis one may need to track these constraints at a finer-grained level, for example which data cache bank, which program block, or which load instruction contributed the most cache miss cycles on the critical path. These different levels of granularity have a multiplicative effect on the amount of in-flight state required for analysis and may increase the complexity accordingly.

We describe an algorithm to manage the large in-flight state required for critical path analysis efficiently. The algorithm trades-off the simulator memory required for maintaining the graph with the cost of traversing the graph. We show how a careful tradeoff can significantly reduce the complexity of the analysis. We present details of the algorithm in Section 4. We apply the analysis on a select set of programs and provide breakdown of the dependence constraints that contribute to the execution critical path. Finally, we show an example where we use this information to improve the performance of a heavily-hand-optimized program by as much as 11%. We present these results in Section 5.

2 TRIPS Architecture

The TRIPS architecture is designed to address key challenges posed by next-generation technologies—power efficiency, high concurrency on a latency-dominated physical substrate, and adaptability to the demands of diverse applications [10, 12]. It uses an EDGE ISA [2], which has two defining characteristics: *block atomic execution* and *direct instruction communication*. The ISA aggregates large groups of instructions into blocks which are logically fetched, executed, and committed as an atomic unit by the hardware. This model amortizes the cost of per-instruction overheads such branch predictions over a large number of instructions. With direct instruction communication, instructions within a block send their results directly to the consumers without writing the value to the register file, enabling lightweight intra-block dataflow execution.

The compiler forms TRIPS blocks, each of which is a predicated hyperblock containing up to 128 instructions [14]. In addition, each block may contain up to 32 *read* and 32 *write* instructions that specify the register inputs and outputs for the block. Further, in all possible executions, a block may execute at most 32 load/store instructions, and produce a constant number of outputs (stores, register writes, and one branch). The compiler also statically determines the placement for all instructions such that they map efficiently onto the microarchitecture [9]. The microarchitecture supports concurrent execution of up to eight blocks, seven of them speculatively. The eight 128-instruction blocks together provide an in-flight window of 1,024 instructions to exploit parallelism.

Figure 1 pictures the prototype TRIPS microarchitecture. It consists of a 4x4 array of execution tiles connected by a routed point-to-point operand transport network. Each execution tile (ET) consists of an integer and floating point unit, a 64-entry reservation station, and is capable of executing one instruction each cycle. The microarchitecture with 16 execution tiles, is thus capable of issuing 16-wide from up to 1,024 instructions in flight. At the periphery of the array are the register file banks (RT) along the top edge and the instruction (IT) and data cache (DT) banks along the left edge. The global control tile (GT) consists of a control-flow predictor, I-cache tag array, and block management state essential for fetch, flush, and commit operations. The RTs, DTs, and the GT are also nodes on the operand network.

Execution of a block proceeds as follows. The GT performs a prediction and obtains the address of the block. It then accesses the I-cache tag array, detects a hit, and sends the cache index in a pipelined fashion to the slave ITs. Each IT independently fetches the instructions and dispatches one instruction per cycle to each ET on the same row (4 total instructions per cycle). An instruction may begin execution in a dataflow fashion as soon as all of its operands are avail-

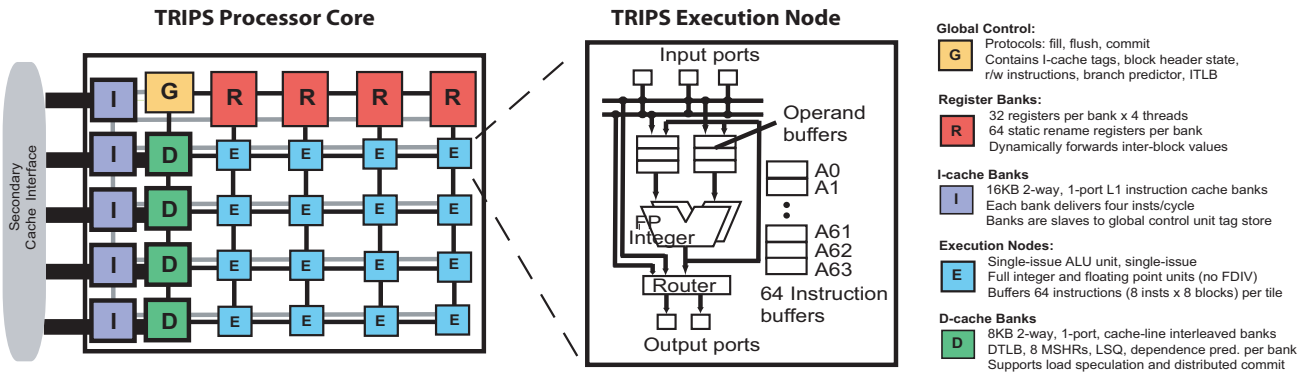

Figure 1. TRIPS prototype microarchitecture

able. Read instructions "execute" by reading their respective block input values and injecting them into the operand network. After an instruction executes, it sends the result to one or more consumer instructions in the same block using the operand network. If the result is a block output, the ET sends it to the appropriate tile—register output to a RT, store to a DT, branch target to the GT. A block completes its execution when all of its outputs have been produced. Once it becomes the oldest in-flight block, the GT sends a commit command to the RTs and the DTs. The commit logic in these tiles first saves the architectural state and then the GT deallocates the block.

All operations during a block's lifetime—fetch, execution, and commit—happen in a fully distributed fashion. Coupled with speculation, this distribution presents challenges for a detailed performance analysis. For example, a typical snapshot of the microarchitectural execution in any given cycle could be as follows: Block 7, the youngest block, is in the middle of its fetch/dispatch process, while some of its instructions have already begun execution. Some instructions in blocks 1–6 are currently executing. An instruction in Block 5 at one of the ETs cannot execute because its issue slot is occupied, while in another ET an instruction cannot send its result out because of contention at the operand network ports. Another instruction is stalled because it is waiting for an operand that is currently in the middle of a long commute from the source instruction. At the same time Block 0, the oldest block, is currently in the process of committing its architectural state, but is stalled due to an unavailable cache port at one of the DTs, even though all of its register outputs have already been committed.

Finding answers to some traditional performance questions becomes a difficult proposition for the TRIPS processor microarchitecture due to its considerable in-flight state—eight blocks and 1,024 instructions, each in one of many states as described above. In addition, the distributed microarchitecture and execution model of TRIPS present some unique questions: a) Do the operand communication latencies limit ILP? b) Are there too many cross-block interactions? c) Should the network bandwidth be doubled? and d) Which network link presents bottlenecks more often? The goal of this paper is not to answer these important questions, but instead provide a powerful framework for analysis that can lead to key insights and answers.

3 Critical Path Model

The critical path model for the TRIPS architecture is heavily based on the dependence-graph model previously developed for superscalar architectures [5]. The model represents various microarchitectural events as nodes in a directed acyclic graph. Edges between the nodes represent dependence constraints among the events. Figure 2 shows a typical dependence graph constructed for a slice of four blocks seen during the program execution. In addition to representing the usual constraints such as data dependences, branch mispredictions, and finite instruction window sizes, the TRIPS model represents constraints imposed by block-atomic execution and operand routing.

Each node in the graph maintains the following information.

- Type of the event: block fetch, operand communication, register read, etc.
- Static delay: statically determinable cycles consumed by an event, e.g., latency of a integer multiply.
- Dynamic delay: latencies introduced by dynamic events, e.g., execution stall cycles due to contention for the issue slot.
- Information about the block and/or instruction associated with the event.
- Information about the tile or network link where the event occurs.

The block-atomic execution model relies on a few global tasks that are performed on behalf of an entire block. These

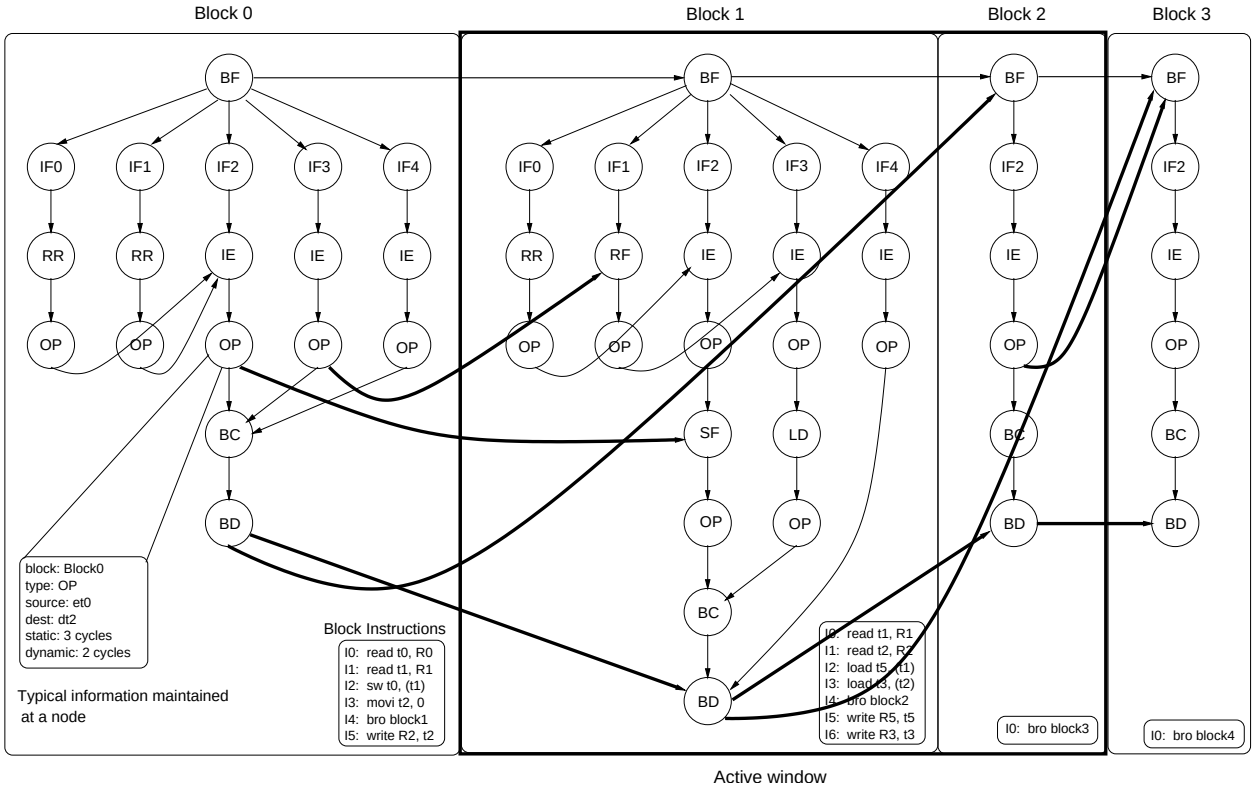

Figure 2. Critical path model for the TRIPS architecture. The example shows the dependence graph for four blocks and a machine window size of two blocks. Inter-block dependences are shown with bold arrows.

tasks introduce dependence constraints for operations not only within the block and but also in other blocks. We summarize these constraints in the following paragraphs and in Table 1.

Intra-Block Fetch Dependences: The control logic at the GT determines the address of the next block to fetch. This event is represented by the graph node BF and denotes the availability of the block's instructions in the cache and the start of the fetch process. The GT takes at least eight cycles to initiate the fetch for the entire block. These eight cycles are recorded as the static latency of the BF event. Any additional latency, for example due to cache misses, is recorded as the dynamic latency of the event.

The delivery of instructions from the I-cache banks to the respective ETs is represented by the IF nodes. The distributed nature of the fetch is represented by the independent IF nodes that are all only dependent on the global block fetch event ($BF \rightarrow IF$). Each IF event has a statically determinable latency based on the location of the tile to which the instruction is dispatched.

Intra-Block Execution Dependences: An instruction executes when all of its operands are ready. Some of these operands may be block inputs that are read from the register banks and delivered on the operand network to the execution tiles. The register read event (RR) represents the read of a block input value from the register bank. It is dependent on the fetch of the corresponding read instruction ($IF \rightarrow RR$). The operand transfer event (OP) represents routing of the value to the tile containing the consumer instruction. The latency for operand routing includes both the statically determinable cycles computed from the number of routing hops and additional cycles resulting from contention for the intermediate network links. The event IE represents the execution of an instruction. After execution, the instruction may route the result to a dependent instruction in the same block. Such data dependences manifest as edges from the execution events of producers to the execution events of consumers via operand communication events ($IE \rightarrow OP \rightarrow IE$).

Intra-Block Commit Dependences: A block completes its execution when all of its outputs — registers, stores and branch target — have been computed. Edges from instruction execution events to the block completion event (BC) via operand communication events represent this dependence ($IE \rightarrow OP \rightarrow BC$). Once a block is known to have completed, the outputs can be committed. This constraint is denoted by the dependence $BC \rightarrow BD$.

Inter-Block Dependences: The fetch of a block can pro-

Name	Events	Dependences	Dependence Edge
BF	Fetch of a block	In-order block fetches	$BF_{i-1} \rightarrow BF_i$
		Recovery from control misprediction	$OP \rightarrow BF$
		Finite window	$BD_{i-w} \rightarrow BF_i$
IF	Fetch of an instruction	Must follow block fetch	$BF \rightarrow IF$
RR	Read of a register	Must follow read instruction fetch	$IF \rightarrow RR$
IE	Instruction execute	Must follow instruction fetch	$IF \rightarrow IE$
		Can execute only after operands have been received	$OP \rightarrow IE$
OP	Operand communication	Can communicate result only after execution has completed	$IE \rightarrow OP$
		Can communicate register values after register read	$RR \rightarrow OP$
BC	Block execution completion	Block completes after all outputs have been produced	$OP \rightarrow BC$
BD	Block commit	Block commits after it completes	$BC \rightarrow BD$
		Blocks must begin their commit operations in-order	$BD_{i-1} \rightarrow BD_i$
RF	Register forward	Register forwarded after value produced	$OP \rightarrow RF$
		Register forwarded after read instruction is fetched	$IF \rightarrow RF$
LD	Load reply	Load reply happens after address is received	$OP \rightarrow LD$
			$LD \rightarrow OP$
SF	Store forward	Forward happens after store value is received	$OP \rightarrow SF$
			$SF \rightarrow OP$

Table 1. Dependences for the TRIPS critical path model

ceed only after the fetch for the previous block has started. This in-order block fetch dependence is represented by the ($BF_{i-1} \rightarrow BF_i$) edges. Similarly, blocks can complete their commit operations only in order. This constraint is represented by $BD_{i-1} \rightarrow BD_i$ edges. Instructions across different blocks could have data dependences through registers. The hardware has the capability to dynamically forward register values from producer instructions in a block to consumers in another block, without waiting for the previous block to commit. This forwarding event is represented by the graph node RF, and associated intra-block fetch dependence $IF \rightarrow RF$ and inter-block dependence $OP \rightarrow RF$.

The hardware supports the execution of up to eight blocks in flight. The fetch of a block can thus proceed only after the deallocation of the eighth block preceding the current one. This dependence is represented by the $BD_{i-w} \rightarrow BF_i$ edges, where w denotes the window size in blocks. Figure 2 depicts a window size of 2 blocks. Finally, branch misprediction in a block constrains the fetch of the successor block. Once the branch instruction is executed and the target communicated to the GT, the fetch process can be initiated. The sequence of dependence edges $IE \rightarrow OP \rightarrow BF$ represents this constraint.

Store-Load Dependences: Load instructions compute the effective addresses at an execution tile and sends them on the network to data cache banks. Data cache banks read the value for loads and route them back to consumer instructions. The cache access is represented by the event LD. Hit latencies appear as static delays and miss latencies appear as dynamic delays. The associated dependences for this sequence of events are represented by the edges $IE \rightarrow OP$, $OP \rightarrow LD$, and $LD \rightarrow OP$. Occasionally, a prior store in the same block or preceding block may have the same address as the load. The load can obtain the correct value only after the store has been received at the data cache bank. Once the store arrives, the load-store queues at the data cache can dynamically forward the value from the store to the matching load. This forwarding event is represented by the node SF.

Critical Path: The longest path in the dependence graph—measured by summing the weights of the nodes in the path—- from the BF event in the first block to the BD event in the last block provides the critical path of execution through the program. By examining the composition of the nodes along the path, one can summarize the contributions of each type of event, each tile or network link in the processor, each program block, or even each instruction in the program to the overall execution of the program. For example, one can determine the critical path cycles resulting from issue slot contention stalls at the tile ET0 while trying to execute the instruction at address `0xdeadf000`. Such information can be fed back to the compiler so that it can find a better placement for the instruction, perhaps by moving it to a different execution tile, and eliminate the contention stall cycles.

4 Computing the Critical Path for TRIPS

The critical path framework for the TRIPS prototype processor consists of two major components: a) a detailed cycle-level simulator, and b) a dependence graph constructor and critical path analyzer. We simulate programs compiled for the TRIPS architecture using a detailed cycle-level simulator. We use traces generated by the simulator to construct the dependence graph of execution. The critical path analyzer then traverses the dependence graph and outputs the critical path information at the desired level of granularity.

The critical path can be computed at different granularities.

- An event-level summary provides the number of cycles spent for each event on the critical path.
- A block-level summary provides the number of cycles for each event in each program block executed on the critical path.
- A tile-level summary provides the contributions of each hardware tile and an instruction-level summary provides the contributions of each program instruction executed on the critical path.

The TRIPS cycle-level simulator faithfully models all the components of the prototype processor shown in Figure 1. It simulates all the tiles, network links, and the pipelines within each tile in great detail. We validated the simulator against a verilog implementation of the prototype and found it to be accurate within 4% on a wide array of microbenchmarks and randomly generated tests. The simulator outputs a trace of the various microarchitectural events that happened during the execution of a program. The trace contains details of each event such as the cycle when it occurred and information about the block or instruction(s) associated with it. We construct the dependence graph using the trace and compute critical paths using the algorithms described in the following sections.

Critical path analysis requires an effective management of the large dependence graph state. Three factors determine the complexity of the algorithm that computes the critical path: a) the size of the graph saved for analysis, b) the number of graph nodes visited during the analysis, and c) the granularity at which the critical path composition is computed. The first factor determines the memory requirements, while the other two determine the computational requirements of the algorithm. In this section, we review two traditional approaches that have opposing requirements on memory and computation. We then present a new algorithm that exploits certain properties of the dependence graph, lowers the requirements on both computation and memory, and delivers the best performance.

4.1 Backward-Pass Algorithm

This algorithm starts at the BD node for the last block. At each step of the algorithm, it visits another node by proceeding to the latest parent node that satisfied the current node's constraints. It terminates at the BF node for the first block. The sequence of the nodes visited is the critical path of execution and by aggregating various information at each of these nodes one can obtain the critical path summaries at different levels of granularity. The advantage of the algorithm is that it does not visit any node that is not on the critical path. However, it requires the entire graph to be constructed and saved before the critical path can be computed. This requirement is clearly intractable for large programs.

4.2 Forward-Pass Algorithm

Prior work on critical path analysis used a simple forward-pass algorithm and saved only a portion of the graph at any given time [5, 18]. The key property of the graph exploited by this algorithm is the fact that no dependence constraint can span more blocks beyond that allowed by the maximum window size of the machine. Consequently, it maintains only the sub-graph of events for a window of $w + 1$ blocks at any given time, where w is the maximum window size. However, each node has to maintain summaries of the critical path in reaching that node.

The critical path summary at each node contains the number of cycles spent for every type of microarchitectural event on the critical path leading to that node. Consequently the cost of copying the summary from one node to another is proportional to the number of different types of events tracked by the tool. The granularity of the critical path composition determines the cost of computing the summaries. If a block-level granularity is desired, the summaries should include the number of critical path cycles for every event in every block. Consequently, the copying costs are proportional to the product of both the number of different blocks executed in the program and the number of different types of events. A tile-level or a instruction-level breakdown has a similar multiplicative effect on the cost of computing the critical path summaries.

The algorithm starts by constructing the graph for the first $w + 1$ blocks. For every node in the first block, it visits all of its successors. During each visit, it propagates all critical path information tracked thus far at a node to the successor. The successor updates its information only if the parent satisfied its constraints the latest. It then adds a new block ($w + 2$, in sequence) to the graph, removes the subgraph corresponding to the first, and repeats the process for the second block.

This algorithm reduces the memory requirements dramatically. However it visits every node in the program's overall graph. In addition, during each visit, a node has to copy the complete critical path breakdowns to its successor. Depending on the required granularity of the breakdowns, these copying costs grow proportionately and for large programs can be prohibitively expensive.

4.3 Mixed Algorithm

The backward-pass algorithm requires copies only along the critical path, but its memory requirements are intractable. By contrast, the forward-pass algorithm keeps

Name	Description	Block counts	Instruction counts	Execution Time (cycles)
a2time01	Misc control code and integer math (EEMBC)	16880	112402	477212
bezier	Bezier curve calculations, fixed-point math (EEMBC)	461694	3807281	2984977
dct8x8	2D discrete cosine transform	40194	3614106	196342
matrix	Matrix multiplication, integer	25624	1355074	230833
sha	NIST secure hash algorithm (MiBench)	15576	1252784	582178

Table 2. Benchmark set used for evaluation

only a small sub-graph in memory, but since it visits every node, its copy requirements can be intractable. A desirable algorithm is one that does not require the entire graph and at the same time does not visit every node to compute the critical path.

A key property of the dependence graph is that for the sub-graph corresponding to an arbitrary window of contiguous blocks, the number of edges from nodes within the window to those outside can always be bounded. This property applies to the dependence graphs for both conventional superscalar and TRIPS architectures. For the TRIPS architecture, these out-going edges can source only a few "output" nodes: a) one BF node, enforcing in-order block fetch start events, b) one BD node, enforcing in-order block commit events, c) one branch communication node for any branch mispredictions, d) one or more register output communication nodes, and e) one or more store communication nodes. The latter two set of nodes can be bounded as they can only belong to the most recently seen eight blocks, each of which can have only up to 32 register writes and 32 stores as permitted by the ISA. We exploit this property in composing an algorithm that uses a combination of both backward and forward passes. One can extend the algorithm fairly easily for a conventional superscalar architecture.

The algorithm maintains the sub-graph of events for a sliding window of $r + 8$ blocks, where r is a large number such that the graph can be feasibly accommodated in memory. The algorithm starts by constructing the graph for the first $r + 8$ blocks. It then does a backward pass starting from each "output" node (in blocks $r - 7$ to r) and collects the critical path information at these nodes. For each output node, it then propagates the critical path information to all its successors similar to the forward pass algorithm. It then removes the top r blocks and adds the next r blocks to the graph. The whole process continues until the critical path information is collected at the commit node for the last block.

Depending on the value for r, the algorithm reduces the number of graph nodes visited and consequently, the number of times the critical path information is copied from one node to another. A large value for r imposes a greater memory requirement for maintaining the in-flight graph state compared to the forward-pass algorithm. But it amortizes that cost by visiting only those nodes that are on the critical path leading to an output node. In our experiments, we found that best setting for r was one that consumed most of the available memory. It must be noted that if r is set to 1, the algorithm is similar to the forward-pass algorithm described above and if set to ∞, it defaults to the backward-pass algorithm.

5 Results

This section shows the results of the critical path analysis on a select set of benchmarks. Our primary goal is to illustrate the type of information that the analysis can provide, and not to demonstrate the raw performance of the architecture. Consequently we limit ourselves to relatively simple programs, based upon real and common algorithms. We used a set of five benchmarks from the following sources—EEMBC [1], MiBench [8], basic math kernels, and basic DSP kernels. Many of these are iterative, repetitive, and have a small enough working set that fit in the level one caches. Table 2 provides a listing of these benchmarks along with the number of blocks and instructions encountered during dynamic execution and the execution time of these programs.

5.1 Algorithm Performance

We first demonstrate the performance of the mixed forward-backward pass algorithm. Figure 3 shows the speed of the critical path framework for different region sizes—the parameter r that determines the number of blocks for which the tool maintains the graph in memory. The x-axis varies the region size and the y-axis shows the analysis time measured in seconds. For these results, the tool computes the critical path at a block-level granularity. For each sample point in the graph, we perform a number of experiments on a dedicated desktop machine and report the average analysis time.

Across all benchmarks, the analysis times improve dramatically as we increase the region sizes from 8 blocks to 64 blocks, at which point the benefits of further increases begin to taper off. At smaller region sizes, the cost of maintaining the graph in memory is insignificant compared to the cost of copying the critical path summaries across different nodes. Higher region sizes increase the memory requirements, but decrease the copying costs. We observe minor

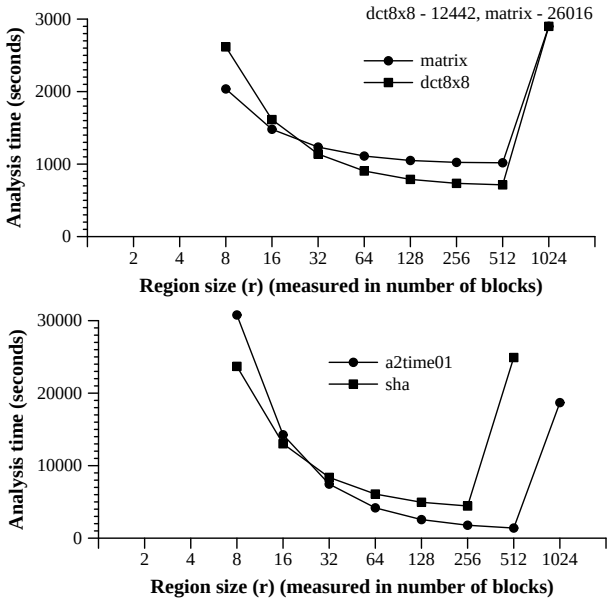

Figure 3. Sensitivity of analysis time to region sizes

improvements for region sizes up to 512 blocks (256 for the benchmark sha). Beyond this size, the memory requirements of the algorithm exceed the capacity of the host machine (1 GB) and the resultant disk swap activity causes a precipitous slowdown in the speed of analysis.

Different benchmarks exhibit different speedups in analysis time. This is because the cost of block-level breakdowns in the critical path summaries is proportional to the number of different program blocks encountered during the execution. Benchmarks $matrix$ and $dct8x8$ contain fewer blocks than benchmarks $a2time01$ and sha. Consequently they exhibit relatively modest improvements of (2x-3x) when varying the region sizes from 8 to 512. On the other hand, the benchmark $a2time01$ exhibits nearly 30x improvement over the same range.

These results show that when computing rich critical path information, the mixed algorithm can deliver orders of magnitude improvements in performance with favorable region sizes. The best performing algorithm is one that just saturates the memory capacity of the host machine.

5.2 Speed of the Critical Path Framework

The speed of the overall critical path framework depends on three components: a) the cycle-level simulator, b) the granularity of the computed critical summaries, and c) the algorithm computing the critical path. Table 3 compares the overhead of computing the critical path at different granularities with the baseline cycle-level simulator. For every benchmark, it shows the simulation speed measured in simulated cycles per seconds for the baseline cycle-level simulator and the relative slowdown of the critical path analysis at four levels of granularity. For this study, we used the mixed forward-backward pass algorithm with a region size of 256.

Granularity	a2time01	bezier01	dct8x8	matrix	sha
base	2.27	1.87	3.30	3.73	3.80
block	4.02	3.67	4.63	5.11	10.93
tile	2.75	2.51	4.22	4.71	7.33
inst	2.90	2.01	8.65	9.56	8.77
cycle-level speed (cycles/s)	1359	1494	1258	1149	1420

Table 3. Relative slowdown of analysis at varying levels of granularity

Computing event-level breakdowns causes the baseline cycle-level simulation to slow down by a factor 1.8x–3.8x across different benchmarks. Adding block-level breakdowns to the analysis causes additional slowdowns of 1.4x–3.8x. The differences in the benchmarks arise from the number of different blocks simulated during the execution. On the other hand, computing the tile-level breakdowns causes a fairly uniform slowdown of about 20%–30% compared to event-level breakdowns. This is because during the backward pass, the constant number of tiles cause a uniform amount of state to be copied from one node to another. The last row shows the cost of computing the contributions of each individual instruction in the most critical program block. The analysis is faster compared to the the block-level analysis for benchmarks $a2time01$ and $bezier01$. This is because these benchmarks have more program blocks than they have instructions in the most critical block, whereas the opposite is true for benchmarks $dct8x8$ and $matrix$.

As shown in Table 3, the speed of critical path analysis can be reduced dramatically depending on the desired granularity of the computation. To keep the analysis tractable, a designer ought to perform critical path analysis, starting with an event-level view and progressively add finer granularities for select portions of the program or hardware resources. For example, if a designer can identify the most critical blocks, s/he can obtain additional information just for that set of blocks with a different simulation.

5.3 Critical Path Breakdown

This section shows the breakdown of the critical path cycles for the benchmark set. Table 4 shows the contribution of each microarchitectural event described in Section 3 for different programs. Naturally, different programs exhibit different behavior. For example, in benchmarks *sha* and *bezier01*, nearly half of the cycles result from raw instruction execution. This shows that even if rest of the bottlenecks were eliminated, the performance of these programs can be improved by 2x at best. Further improvements must come from reductions in the instruction counts through code

Event	a2time01	bezier01	dct8x8	matrix_1	sha
BC	1.3%	0.9%	1.5%	1.6%	0.4%
BD	5.0%	2.7%	3.3%	4.6%	1.2%
BF	21.8%	6.3%	31.8%	7.0%	8.2%
IE	24.7%	45.4%	18.7%	24.6%	60.3%
IF	4.2%	2.3%	5.2%	8.1%	0.5%
LD	16.9%	13.7%	0.6%	2.6%	0.2%
OP	21.8%	25.1%	34.4%	38.0%	25.9%
RR	3.6%	0.2%	0.3%	6.4%	0.1%
RF	0.8%	3.4%	4.1%	7.2%	3.0%
SF	0.0%	0.0%	0.0%	0.0%	0.0%
Total (cycles)	477212	2984977	196342	230833	582178

Block Name	(Static,	Dynamic)	Total Delay (cycles)
matrix_mult$2	(74352,	56216)	130568
matrix_check$1	(26047,	31440)	57487
matrix_mult$1	(23969,	13785)	37754
main$4	(1812,	236)	2048
matrix_check	(1802,	239)	2041

a) Block-level breakdown for the program matrix

Instruction	(Static,	Dynamic)	Total Delay
addi	(3496,	4428)	7924
mul	(3285,	1965)	5250
mul	(2380,	1777)	4157
add	(1985,	777)	2762
tlti_f	(1552,	886)	2438

b) Instruction-level breakdown for block matrix_mult$2 in matrix

Table 4. Composition of the critical path for different programs

optimizations and reductions in the hardware latency of the arithmetic units.

A significant portion of the critical path cycles across all benchmarks can be attributed to the operand communication latencies. Static components of the latency correspond to the number of routing hops. Dynamic components of the latency result from network link contention. Both of these components can be reduced with better placements for the critical instructions. By obtaining a block-level or instruction-level breakdown for the critical path, one can identify the program blocks and instructions contributing to the critical latencies and focus the scheduling policies towards them. Table 4 shows an example of these breakdowns for the benchmark $matrix$. It shows the critical path contributions for the top five program blocks and the top five instructions in the block $matrix\_mult\$2$.

5.4 Improving Performance Using Critical Path Analysis

In this section, we show the utility of critical path analysis in improving the performance of an application. We use the program $memset$ for this exercise. This program is a C library routine that sets a range of bytes in memory to a given value. Table 5 shows the breakdown of the critical path cycles for two versions of the program. The baseline program is a hand-tuned version of $memset$. The optimizations performed by hand include aggressive hyperblock formation using loop unrolling and predication, and hand placement of instructions. These optimizations improve the performance of $memset$ by over 8x relative to automatically compiled code.

From Table 5 we observe that nearly 70% of the critical path cycles in the baseline program were consumed by operand communication and instruction execute events. We further observe that a large fraction of these cycles are dynamic delays, which indicate contention stall cycles at the

Event	Baseline			Optimized		
	SD	DD	TD	SD	DD	TD
BF	2728	12533	15261	4896	13144	18040
BC	0	181	181	0	1967	1967
BD	408	507	915	3968	5262	9230
OP	18730	17777	36507	9249	5839	15088
IE	9712	15554	25266	3871	1826	5697
RR	92	23	115	528	14314	14842
SF	0	0	0	0	0	0
RF	7678	79	7757	2	0	2
IF	2521	0	2521	5351	0	5351
LD	246	542	788	244	542	786
Total	42115	47196	89311	28109	42894	71003

Table 5. Overall critical path breakdown for two versions of $memset$. The label SD denotes the static delay, DD denotes the dynamic delay, and TD denotes the total delay for an event.

operand network links and at the execution tile issue slots. To identify the specific program block causing these contention cycles, we re-performed the analysis to track the critical path composition on a per-block basis. We observed that nearly 70% of the critical path cycles resulted from events in one program block $memset\_test\$6$. Table 6 shows these results. Nearly 90% of all operand communication and instruction execute latencies in the program's overall critical path result from this block.

Instructions in the block $memset\_test\$6$ are in one of four categories: $store$ instructions, $move$ instructions to distribute the base address for the stores, $move$ instructions to distribute the data for the stores, and loop induction instructions. To identify specific instructions that cause bottlenecks, we re-ran the analysis to obtain critical path breakdowns for each instruction in the block $memset\_test\$6$. Table 7 shows the contribution of the top five instructions in that block to the critical operand communication and instruction execution latencies. We observe that nearly 50% of these cycles resulted from just one single instruction. Examining the tile placements in the schedule, we observed

Event	Baseline			Optimized		
	SD	DD	TD	SD	DD	TD
OP	15650	17276	32926	5755	4310	10065
IE	7733	15209	22942	2120	1181	3301
RF	7548	0	7548	0	0	0
RR	0	0	0	400	12355	12755
SF	0	0	0	0	0	0
IF	114	0	114	2312	0	2312
LD	0	0	0	0	0	0
BF	264	61	325	2432	580	3012
BC	0	129	129	0	1485	1485
BD	312	429	741	3200	4400	7600
Total (cycles)	31621	33104	64725	16219	24311	40530

Table 6. Critical path composition for the block $memset\_test\$6$ in the program $memset$

that this instruction, a store, was obtaining its base address from a move instruction placed at a different execution tile. In fact, all the consuming stores of this move instruction were placed at a different tile. This introduced one cycle of operand communication latency between the issue of the move and the target store instructions. To remove this latency, we re-adjusted the schedules by placing the move instruction and all of its target stores at the same tile.

The results for the optimized version of $memset\_test\$6$ are shown in Tables 5, 6, and 7 under the label $Optimized$. We observe that the overall performance improved by nearly 11%. As expected, we observe a significant reduction in operand communication and instruction execute latencies on the program's overall critical path. The critical path contribution of the top-most block, still $memset\_test\$6$, decreased by a greater fraction than the overall execution time. This behavior occurs because portions of the execution paths through this block are no longer critical compared to concurrent paths through other blocks. Table 5 also shows significant increases in the contributions of the block fetch, block completion, and block commit operations. The block $memset\_test\$6$ also exhibits similar sharp increases in contributions of other events. Reducing the effect of operand communication and instruction execution bottlenecks, expose these new bottlenecks which are candidates for future optimizations.

Finally, we note that we embarked on this hand-optimization exercise to improve the performance of library routines over and beyond what is delivered by the current TRIPS compiler. The compiler's scheduler can exploit this technique to automatically improve the placement of the instructions. Simulated annealing is one technique to arrive at near-optimal placements. Critical path analysis can be used by the annealer to prioritize for the critical instructions and help it converge to a solution faster. We are exploring this technique in current work.

Baseline			Optimized		
SD	DD	TD	SD	DD	TD
7506	25288	32794	1172	631	1803
7484	819	8303	879	101	980
95	304	399	424	473	897
93	156	249	289	578	867
128	85	213	586	71	657

Table 7. Operand communication and instruction execution cycles on the critical path for the top five instructions in the block $memset\_test\$6$ in the program $memset$

6 Related Work

The notion of critical path models for processor architectures is not new. Prior research has focused on one of the following: identifying criticality of specific classes of instruction, critical path modeling, critical path prediction and optimizations to improve performance.

Early research on critical path has generally focussed on understanding the performance of load instructions. Srinivasan et al. quantify a measure of load criticality called latency tolerance [16]. In subsequent work, Srinivasan et al. propose a heuristics-based hardware predictor of critical loads and quantitatively compare criticality-based prefetching techniques with locality-based techniques [15]. Other researchers have also proposed hardware estimators of critical loads and provided techniques to improve cache performance of critical loads at the expense of non-critical ones [7, 11]. All of these approaches are specific to load instructions and cannot be easily extended to other instructions or microarchitectural resources.

Our work is closest to the dependence graph-based models of the critical path developed by Fields et al. [5] and Tune et al. [18]. As explained before, these models abstract the execution of a program with a dependence graph. By manipulating the graph in different ways, they show how different measures of criticality — slack [4], tautness [18] and interaction costs [6] — can be computed efficiently with a simulator. Tune et al. [17, 18] and Fields et al. [5] use these models to develop run-time critical path predictors. In a later work, Fields et al. show how the model can be used to gain insights about secondary critical paths [6]. Our research extends these models for the TRIPS architecture.

In later work, researchers have developed techniques to predict the criticality of all types of instructions [3, 5, 13, 17]. These techniques provide critical path measures to varying degrees of accuracy and have been applied to improve value prediction performance, clustered architecture scheduling, and power consumption.

7 Conclusions

Critical path models provide a powerful framework for drawing key insights into the performance of a processor.

Researchers have developed a plethora of techniques to exploit these models for improving application performance. All of these techniques presuppose a conventional superscalar processor as the underlying hardware substrate. In this paper, we develop a functional critical path model for the TRIPS architecture.

We show how the unique constraints in the architecture, specifically the block-atomic execution model and operand network links, can be modeled with the dependence graph. The wider issue and larger instruction window in the TRIPS processor increase the complexity of the critical path analysis considerably. Computing the critical path information at different levels of granularity also imposes different demands on the computational and memory requirements for the analysis. We develop an algorithm that, without any loss in precision, can significantly reduce the complexity of the critical path analysis by up to 30x compared to techniques used in prior work. We demonstrate the functionality of this framework and show critical path summaries for a sample set of programs. Guided by the analysis, we improve the performance of a previously hand-optimized program by 11%.

Our current framework provides the composition of the critical path which is not always an accurate indicator of the true bottlenecks in an architecture. By extending the framework, one can obtain several other metrics of criticality such as tautness and interaction costs to provide better indicators of bottlenecks. Our experience with the critical path framework has been that it has taken the tedium out of hand-optimizing certain programs. It is conceivable that the compiler's scheduler can be augmented to use this framework, automatically identify the potential bottlenecks, and produce schedules to minimize their impact. With the extensions described above, we expect the framework to offer additional insights into the bottlenecks in this class of architectures.

Acknowledgments

We thank the anonymous reviewers for their suggestions that helped improve the quality of this paper. This research is supported by the Defense Advanced Research Projects Agency under contracts F33615-01-C-4106 and NBCH30390004 and an NSF instrumentation grant EIA-0303609.

References

[1] EEMBC: The embedded microprocessor benchmark consortium. http://www.eembc.org.

[2] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team. Scaling to the End of Silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[3] B. Calder, G. Reinman, and D. M. Tullsen. Selective value prediction. In *Proceedings of the 26th Annual International Symposium on Computer architecture*, pages 64–74, May 1999.

[4] B. Fields, R. Bodik, and M. D. Hill. Slack: Maximizing performance under technological constraints. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 47–58, June 2002.

[5] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 74–85, July 2001.

[6] B. A. Fields, R. Bodik, M. D. Hill, and C. J. Newburn. Using interaction costs for microarchitectural bottleneck analysis. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 228–239, December 2003.

[7] B. R. Fisk and R. I. Bahar. The non-critical buffer: Using load latency tolerance to improve data cache efficiency. In *Proceedings of the 1999 IEEE International Conference on Computer Design*, pages 538–545, October 1999.

[8] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *IEEE 4th Annual Workshop on Workload Characterization*, pages 3–14, December 2001.

[9] R. Nagarajan, S. K. Kushwaha, D. Burger, K. S. McKinley, C. Lin, and S. W. Keckler. Static Placement, Dynamic Issue (SPDI) Scheduling for EDGE Architectures. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques*, pages 74–84, October 2004.

[10] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 40–51, December 2001.

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

[12] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, June 2003.

[13] J. S. Seng, E. S. Tune, and D. M. Tullsen. Reducing power with dynamic critical path information. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 114–123, December 2001.

[14] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley. Compiling for EDGE architectures. In *Fourth International ACM/IEEE Symposium on Code Generation and Optimization (CGO)*, March 2006.

[15] S. T. Srinivasan, R. D. ching Ju, A. R. Lebeck, and C. Wilkerson. Locality vs. Criticality. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 132–143, July 2001.

[16] S. T. Srinivasan and A. R. Lebeck. Load latency tolerance in dynamically scheduled processors. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 148–159, November 1998.

[17] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Dynamic prediction of critical path instructions. In *Proceedings of the Seventh International Symposium on High Performance Computer Architecture*, pages 185–195, January 2001.

[18] E. Tune, D. M. Tullsen, and B. Calder. Quantifying instruction criticality. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques*, pages 104–113, September 2002.