Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors

Onur Mutlu § Jared Stark † Chris Wilkerson ‡ Yale N. Patt §

§ECE Department
The University of Texas at Austin
{onur,patt}@ece.utexas.edu

†Microprocessor Research
Intel Labs
jared.w.stark@intel.com

‡Desktop Platforms Group
Intel Corporation
chris.wilkerson@intel.com

Abstract

Today's high performance processors tolerate long latency operations by means of out-of-order execution. However, as latencies increase, the size of the instruction window must increase even faster if we are to continue to tolerate these latencies. We have already reached the point where the size of an instruction window that can handle these latencies is prohibitively large, in terms of both design complexity and power consumption. And, the problem is getting worse. This paper proposes runahead execution as an effective way to increase memory latency tolerance in an out-of-order processor, without requiring an unreasonably large instruction window. Runahead execution unblocks the instruction window blocked by long latency operations allowing the processor to execute far ahead in the program path. This results in data being prefetched into caches long before it is needed. On a machine model based on the Intel® Pentium® 4 processor, having a 128-entry instruction window, adding runahead execution improves the IPC (Instructions Per Cycle) by 22% across a wide range of memory intensive applications. Also, for the same machine model, runahead execution combined with a 128-entry window performs within 1% of a machine with no runahead execution and a 384-entry instruction window.

1. Introduction

Today's high performance processors tolerate long latency operations by implementing out-of-order instruction execution. An out-of-order execution engine tolerates long latencies by moving the long-latency operation "out of the way" of the operations that come later in the instruction stream and that do not depend on it. To accomplish this, the processor buffers the operations in an instruction window, the size of which determines the amount of latency the out-of-order engine can tolerate.

Today's processors are facing increasingly larger latencies. With the growing disparity between processor and memory speeds, operations that cause cache misses out to main memory take hundreds of processor cycles to complete execution [25]. Tolerating these latencies solely with out-of-order execution has become difficult, as it requires ever-larger instruction windows, which increases design complexity and power consumption. For this reason, computer architects developed software and hardware prefetching methods to tolerate these long memory latencies.

We propose using runahead execution [10] as a substitute for building large instruction windows to tolerate very long latency operations. Instead of moving the long-latency operation "out of the way," which requires buffering it and the instructions that follow it in the instruction window, runahead execution on an out-of-order execution processor tosses it out of the instruction window.

When the instruction window is blocked by the long-latency operation, the state of the architectural register file is checkpointed. The processor then enters "runahead mode." It distributes a bogus result for the blocking operation and tosses it out of the instruction window. The instructions following the blocking operation are fetched, executed, and pseudo-retired from the instruction window. By pseudo-retire, we mean that the instructions are executed and completed as in the conventional sense, except that they do not update architectural state. When the blocking operation completes, the processor re-enters "normal mode." It restores the checkpointed state and refetches and re-executes instructions starting with the blocking operation.

Runahead's benefit comes from transforming a small instruction window which is blocked by long-latency operations into a non-blocking window, giving it the performance of a much larger window. The instructions fetched and executed during runahead mode create very accurate prefetches

Intel® and Pentium® are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

for the data and instruction caches. These benefits come at a modest hardware cost, which we will describe later.

In this paper we only evaluate runahead for memory operations that miss in the second-level cache, although it can be initiated on any long-latency operation that blocks the instruction window. We use Intel's IA-32 ISA, and throughout this paper, microarchitectural parameters (e. g., instruction window size) and IPC (Instructions Per Cycle) performance are reported in terms of micro-operations. Using a machine model based on the Intel Pentium 4 processor, which has a 128-entry instruction window, we first show that current out-of-order execution engines are unable to tolerate long main memory latencies. Then we show that runahead execution can better tolerate these latencies and achieve the performance of a machine with a much larger instruction window. Our results show that a baseline machine with a realistic memory latency has an IPC performance of 0.52, whereas a machine with a 100% second-level cache hit ratio has an IPC of 1.26. Adding runahead increases the baseline's IPC by 22% to 0.64, which is within 1% of the IPC of an identical machine with a 384-entry instruction window.

2. Related work

Memory access is a very important long-latency operation that has concerned researchers for a long time. Caches [29] tolerate memory latency by exploiting the temporal and spatial reference locality of applications. Kroft [19] improved the latency tolerance of caches by allowing them to handle multiple outstanding misses and to service cache hits in the presence of pending misses.

Software prefetching techniques [5, 22, 24] are effective for applications where the compiler can statically predict which memory references will cause cache misses. For many applications this is not a trivial task. These techniques also insert prefetch instructions into applications, increasing instruction bandwidth requirements.

Hardware prefetching techniques [2, 9, 16, 17] use dynamic information to predict what and when to prefetch. They do not require any instruction bandwidth. Different prefetch algorithms cover different types of access patterns. The main problem with hardware prefetching is the hardware cost and complexity of a prefetcher that can cover the different types of access patterns. Also, if the accuracy of the hardware prefetcher is low, cache pollution and unnecessary bandwidth consumption degrades performance.

Thread-based prefetching techniques [8, 21, 31] use idle thread contexts on a multithreaded processor to run threads that help the primary thread [6]. These helper threads execute code which prefetches for the primary thread. The main disadvantage of these techniques is they require idle thread contexts and spare resources (e. g., fetch and execution bandwidth), which are not available when the processor is well used.

Runahead execution [10] was first proposed and evaluated as a method to improve the data cache performance of a five-stage pipelined in-order execution machine. It was shown to be effective at tolerating first-level data cache and instruction cache misses [10, 11]. In-order execution is unable to tolerate any cache misses, whereas out-of-order execution can tolerate some cache miss latency by executing instructions that are independent of the miss. We will show that out-of-order execution cannot tolerate long-latency memory operations without a large, expensive instruction window, and that runahead is an alternative to a large window. We also introduce the "runahead cache" to effectively handle store-load communication during runahead mode.

Balasubramonian et al. [3] proposed a mechanism to execute future instructions when a long-latency instruction blocks retirement. Their mechanism dynamically allocates a portion of the register file to a "future thread," which is launched when the "primary thread" stalls. This mechanism requires partial hardware support for two different contexts. Unfortunately, when the resources are partitioned between the two threads, neither thread can make use of the machine's full resources, which decreases the future thread's benefit and increases the primary thread's stalls. In runahead execution, both normal and runahead mode can make use of the machine's full resources, which helps the machine to get further ahead during runahead mode.

Finally, Lebeck et al. [20] proposed that instructions dependent on a long-latency operation be removed from the (relatively small) scheduling window and placed into a (relatively big) waiting instruction buffer (WIB) until the operation is complete, at which point the instructions are moved back into the scheduling window. This combines the latency tolerance benefit of a large instruction window with the fast cycle time benefit of a small scheduling window. However, it still requires a large instruction window (and a large physical register file), with its associated cost.

3. Out-of-order execution and memory latency tolerance

3.1. Instruction and scheduling windows

Out-of-order execution can tolerate cache misses better than in-order execution by scheduling operations that are independent of the miss. An out-of-order execution machine accomplishes this using two windows: the instruction window and the scheduling window. The instruction window holds all the instructions that have been decoded but not yet committed to the architectural state. Its main purpose is to guarantee in-order retirement of instructions in order to support precise exceptions. The scheduling window holds a subset of the instructions in the instruction window. Its main purpose is to search its instructions each cycle for those that are ready to execute and to schedule them for execution.

A long-latency operation blocks the instruction window until it is completed. Although later instructions may have completed execution, they cannot retire from the instruction window. If the latency of the operation is long enough and the instruction window is not large enough, instructions pile up in the instruction window and it becomes full. The machine then stalls and stops making forward progress. [1]

3.2. Memory latency tolerance of an out-of-order processor

In this section, we show that an idealized version of a current out-of-order execution machine spends most of its time stalling, mostly waiting for main memory. We model both a 128 and a 2048 entry instruction window. All other machine buffers except the scheduler are set to either 128 or 2048 entries so they do not create bottlenecks. The fetch engine is ideal in that it never suffers from cache misses and always supplies a fetch-width's worth of instructions every cycle. Thus fetch never stalls. However, it does use a real branch predictor. The other machine parameters are the same as those of the current baseline—which is based on the Intel Pentium 4 processor—and are shown in Table 2.

Figure 1 shows the percentage of cycles the instruction window is stalled for seven different machines. The number on top of each bar is the IPC of the machine. The data is an average over all benchmarks simulated. (See Section 5.)

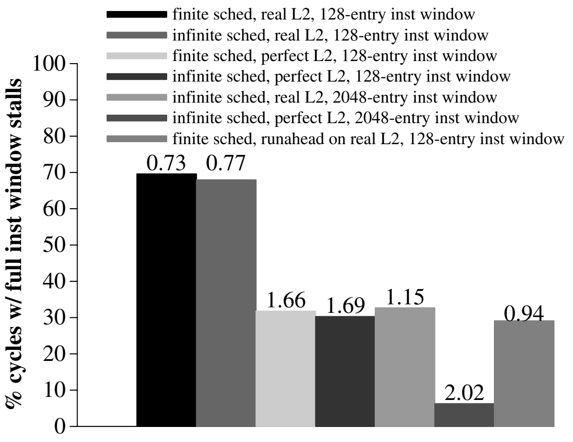

Figure 1. Percentage of cycles with full instruction window stalls. The number on top of each bar is the IPC of the machine.

[1]The machine can still fetch and buffer instructions but it cannot decode, schedule, execute, and retire them.

The machine with a 128-entry instruction window, a finite scheduling window size, and a real L2 cache spends 71% of its cycles in full instruction window stalls, where no progress is made. If the scheduler is removed from being a bottleneck by making the scheduling window size infinite, the instruction window still remains a bottleneck, with 70% of all cycles spent in full window stalls. If instead the main memory latency is removed from being a bottleneck by making the L2 cache perfect, the machine only wastes 32% of its cycles in full window stalls. Thus most of the stalls are due to main memory latency. Eliminating this latency increases the IPC by 143%.

The machine with a 2048-entry instruction window and a real L2 cache is able to tolerate main memory latency much better than the machines with 128-entry instruction windows and real L2 caches. Its percentage of full window stalls is similar to those of the machines with 128-entry instruction windows and perfect L2 caches. However, its IPC is not as high, because L2 misses are still not free. The machine with a 2048-entry instruction window and a perfect L2 cache is shown for reference. It has the highest IPC and almost no full window stalls. As shown in the rightmost bar, runahead execution eliminates most of the full window stalls due to main memory latency on a 128-entry window machine, increasing IPC by more than 20%.

3.3. The insight behind runahead

The processor is unable to make progress while the instruction window is blocked waiting for main memory. Runahead execution removes the blocking instruction from the window, fetches the instructions that follow it, and executes those that are independent of it. Runahead's performance benefit comes from fetching instructions into the fetch engine's caches and executing the independent loads and stores that miss the first or second level caches. All these cache misses are serviced in parallel with the miss to main memory that initiated runahead mode, and provide useful prefetch requests. The premise is that this nonblocking mechanism lets the processor fetch and execute many more useful instructions than the instruction window normally permits. If this is not the case, runahead provides no performance benefit over out-of-order execution.

4. Implementation of runahead execution in an out-of-order processor

In this section, we describe the implementation of runahead execution on an out-of-order processor, where instructions access the register file after they are scheduled and before they execute. The Intel Pentium 4 processor [13], MIPS* R10000* microprocessor [30], and

*Other names and brands may be claimed as the property of others.

Compaq* Alpha* 21264 processor [18] are examples of such a microarchitecture. In some other microarchitectures, such as the Intel Pentium Pro processor [12], instructions access the register file before they are placed in the scheduler. The implementation details of runahead are slightly different between the two microarchitectures, but the basic mechanism works the same way.

Figure 2 shows the simulated processor pipeline. Dashed lines in this figure show the flow of miss traffic out of the caches. Shaded structures constitute the hardware required to support runahead execution and will be explained in this section. The Frontend Register Alias Table (RAT) [13] is used for renaming incoming instructions and contains the speculative mapping of architectural registers to physical registers. The Retirement RAT [13] contains pointers to those physical registers that contain committed architectural values. It is used for recovery of state after branch mispredictions and exceptions. Important machine parameters are given in Section 5.

4.1. Entering runahead mode

A processor can enter runahead mode at any time. A data cache miss, an instruction cache miss, and a scheduling window stall are only a few of many possible events that can trigger a transition into runahead mode. In our implementation, the processor enters runahead mode when a memory operation misses in the second-level cache and that memory operation reaches the head of the instruction window. The address of the instruction that causes entry into runahead mode is recorded. To correctly recover the architectural state on exit from runahead mode, the processor checkpoints the state of the architectural register file. For performance reasons, the processor also checkpoints the state of the branch history register and the return address stack. All instructions in the instruction window are marked as "runahead operations" and will be treated differently by the microarchitecture. Any instruction that is fetched in runahead mode is also marked as a runahead operation.

Checkpointing of the architectural register file can be accomplished by copying the contents of the physical registers pointed to by the Retirement RAT, which may take time. To avoid performance loss due to copying, the processor can always update the checkpointed architectural register file during normal mode. When a non-runahead instruction retires from the instruction window, it updates its architectural destination register in the checkpointed register file with its result. No cycles are lost for checkpointing. Other checkpointing mechanisms can also be used, but their discussion is beyond the scope of this paper. No updates to the checkpointed register file are allowed during runahead mode.

It is worthwhile to note that, with this kind of implementation, runahead execution introduces a second level of checkpointing mechanism to the pipeline. Although the Retirement RAT points to the architectural register state in normal mode, it points to the pseudo-architectural register state during runahead mode and reflects the state updated by pseudo-retired instructions.

4.2. Execution in runahead mode

The main complexities involved with execution of runahead instructions are with memory communication and propagation of invalid results. Here we describe the rules of the machine and the hardware required to support them.

Invalid bits and instructions. Each physical register has an invalid (INV) bit associated with it to indicate whether or not it has a bogus value. Any instruction that sources a register whose invalid bit is set is an invalid instruction. INV bits are used to prevent bogus prefetches and resolution of branches using bogus data.

If a store instruction is invalid, it introduces an INV value to the memory image during runahead. To handle the communication of data values (and INV values) through memory during runahead mode, we use a small "runahead cache" which is accessed in parallel with the first-level data cache. We describe the rationale behind the runahead cache and its design later in this section.

Propagation of INV values. The first instruction that introduces an INV value is the instruction that causes the processor to enter runahead mode. If this instruction is a load, it marks its physical destination register as INV. If it is a store, it allocates a line in the runahead cache and marks its destination bytes as INV.

Any invalid instruction that writes to a register marks that register as INV after it is scheduled or executed. Any valid operation that writes to a register resets the INV bit of its destination register.

Runahead store operations and runahead cache. In previous work [10], runahead store instructions do not write their results anywhere. Therefore, runahead loads that are dependent on valid runahead stores are regarded as invalid instructions and dropped. In our experiments, partly due to the limited number of registers in the IA-32 ISA, we found that forwarding the results of runahead stores to runahead loads is essential for high performance (See Section 6.4).

If both the store and its dependent load are in the instruction window, this forwarding is accomplished through the store buffer that already exists in current out-of-order processors. However, if a runahead load depends on a runahead store that has already pseudo-retired (which means that the store is no longer in the store buffer), it should get the result of the store from some other location. One possibil-

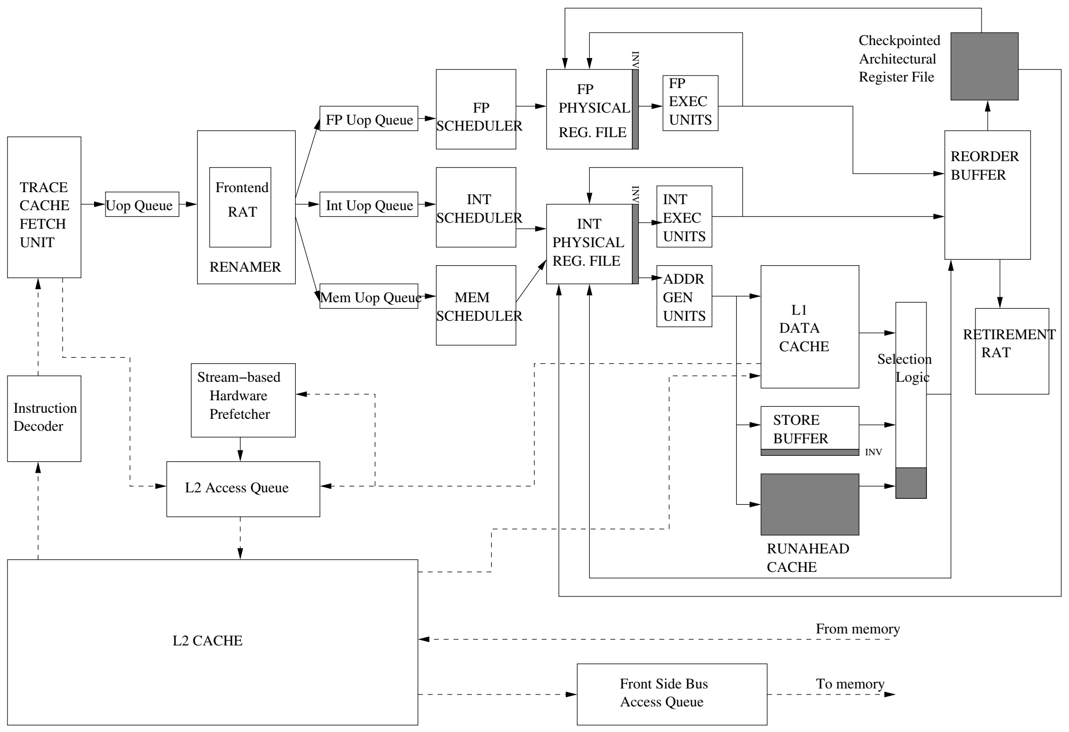

Figure 2. Processor model used for description and evaluation of runahead. Figure is not to scale.

ity is to write the result of the pseudo-retired store into the data cache. This introduces extra complexity to the design of the data cache (and possibly to the second-level cache), because the data cache needs to be modified so that data written by speculative runahead stores is not used by future non-runahead instructions. Writing the data of speculative stores into the data cache can also evict useful cache lines. Another possibility is to have a large fully-associative buffer that stores the results of pseudo-retired runahead store instructions. But, the size and access time of this associative structure can be prohibitively large. Also, such a structure cannot handle the case where a load depends on multiple stores, without increased complexity.

As a simpler alternative, we propose using the runahead cache to hold the results and INV status of the pseudo-retired runahead stores. The runahead cache is addressed just like the data cache, but it can be much smaller in size, because only a small number of store instructions pseudo-retire during runahead mode.

Although, we call it a cache (because it is physically the same structure as a traditional cache), the purpose of the runahead cache, is not to "cache" data. Its purpose is to provide communication of data and INV status between instructions. The evicted cache lines are not stored back in any other larger storage, they are simply dropped. The runahead cache is only accessed by runahead loads and stores. In normal mode, no instruction accesses it.

To support correct communication of INV bits between stores and loads, each entry in the store buffer and each byte in the runahead cache has a corresponding INV bit. Each byte in the runahead cache also has another bit associated with it (the STO bit) indicating whether or not a store has written to that byte. An access to the runahead cache results in a hit only if the accessed byte was written by a store (STO bit is set) and the accessed runahead cache line is valid. The runahead stores follow the following rules to update these INV and STO bits and store their results:

1. When a valid runahead store completes execution, it writes its data into its store buffer entry (just like in a normal processor) and resets the associated INV bit of the entry. In the meantime, it queries the data cache and sends a prefetch request down the memory hierarchy if it misses in the data cache.
2. When an invalid runahead store is scheduled, it sets the INV bit of its associated store buffer entry.
3. When a valid runahead store exits the instruction window, it writes its result into the runahead cache, and resets the INV bits of the written bytes. It also sets the STO bits of the bytes it writes to.
4. When an invalid runahead store exits the instruction window, it sets the INV bits and the STO bits of the bytes it writes into (if its address is valid).
5. Runahead stores never write their results into the data cache.

One complication arises when the address of a store operation is invalid. In this case, the store operation is simply treated as a NOP. Since loads are unable to identify their dependencies on such stores, it is likely they will incorrectly load a stale value from memory. This problem can be mitigated through the use of memory dependence predictors [7, 23] to identify the dependence between an INV-address store and its dependent load. Once the dependence has been identified, the load can be marked INV if the data value of the store is INV. If the data value of the store is valid, it can be forwarded to the load.

Runahead load operations. A runahead load operation can be invalid due to three different reasons:

1. It may source an INV physical register.
2. It may be dependent on a store that is marked as INV in the store buffer.
3. It may be dependent on a store that has already pseudo-retired and was INV.

The last case is detected using the runahead cache. When a valid load executes, it accesses three structures in parallel: the data cache, the runahead cache, and the store buffer. If it hits in the store buffer and the entry it hits is marked valid, the load gets its data from the store buffer. If the load hits in the store buffer and the entry is marked INV, the load marks its physical destination register as INV.

A load is considered to hit in the runahead cache only if the cache line it accesses is valid and STO bit of any of the bytes it accesses in the cache line is set. If the load misses in the store buffer and hits in the runahead cache, it checks the INV bits of the bytes it is accessing in the runahead cache. The load executes with the data in the runahead cache if none of the INV bits is set. If any of the sourced data bytes is marked INV, then the load marks its destination INV.

If the load misses in both the store buffer and runahead cache, but hits in the data cache, then it uses the value from the data cache and is considered valid. Nevertheless, it may actually be invalid because of two reasons: 1) it may be dependent on a store with INV address, or 2) it may be dependent on an INV store which marked its destination bytes in the runahead cache as INV, but the corresponding line in the runahead cache was deallocated due to a conflict. However, both of these are rare cases which do not affect performance significantly.

If the load misses in all three structures, it sends a request to the second-level cache to fetch its data. If this request hits in the second-level cache, data is transferred from the second-level cache to the first-level cache and load completes its execution. If the request misses in the second-level cache, the load marks its destination register as INV and is removed from the scheduler, just like the load that caused entry into runahead mode. The request is sent to memory like a normal load request that misses the L2 cache.

Execution and prediction of branches. Branches are predicted and resolved in runahead mode exactly the same way they are in normal mode except for one difference: A branch with an INV source, like all branches, is predicted and updates the global branch history register speculatively, but, unlike other branches, it can never be resolved. This is not a problem if the branch is correctly predicted. However, if the branch is mispredicted, the processor will always be on the wrong path after the fetch of this branch until it hits a control-flow independent point. We call the point in the program where a mispredicted INV branch is fetched the "divergence point." Existence of divergence points is not necessarily bad for performance, but as we will show later, the later they occur in runahead mode, the better the performance improvement.

One interesting issue with branch prediction is the training policy of the branch predictor tables during runahead mode. One option—and the option we use for our implementation—is to always train the branch predictor tables. If a branch executes in runahead mode first and then in normal mode, such a policy results in branch predictor being trained twice by the same branch. Hence, the predictor tables are strengthened and the counters may lose their hysteresis. A second option is to never train the branch predictor in runahead mode. This results in lower branch prediction accuracy in runahead mode, which degrades performance and moves the divergence point closer in time to runahead entry point. A third option is to always train the branch predictor in runahead mode, but also to use a queue to communicate the results of branches from runahead mode to normal mode. The branches in normal mode are predicted using the predictions in this queue, if a prediction exists. If a branch is predicted using a prediction from the queue, it does not train the predictor tables again. A fourth option is to use two separate predictor tables for runahead mode and normal mode and to copy the table information from normal mode to runahead mode on runahead entry. This option is costly to implement in hardware but we simulated it to determine how much the twice-training policy of the first option matters. Our results show that training the branch predictor table entries twice does not show significant performance loss compared to the fourth option.

Instruction pseudo-retirement during runahead mode. During runahead mode, instructions leave the instruction window in program order. If an instruction reaches the head of the instruction window it is considered for pseudo-retirement. If the instruction considered for pseudo-retirement is INV, it is moved out of the window immediately. If it is valid, it needs to wait until it is executed (at which point it may become INV) and its result is written into the physical register file. Upon pseudo-retirement, an instruction releases all resources allocated for its execution.

Both valid and invalid instructions update the Retirement RAT when they leave the instruction window. The Retirement RAT does not need to store INV bits associated with each register, because physical registers already have INV bits associated with them.

4.3. Exiting runahead mode

An exit from runahead mode can be initiated at any time. For simplicity, we handle the exit from runahead mode the same way a branch misprediction is handled. All instructions in the machine are flushed and their buffers are deallocated. The checkpointed architectural register file is copied into a pre-determined portion of the physical register file. The frontend and retirement RATs are also repaired so that they point to the physical registers that hold the values of architectural registers. This recovery is accomplished by reloading the same hard-coded mapping into both of the alias tables. All lines in the runahead cache are invalidated (and STO bits set to 0) and the checkpointed branch history register and return address stack are restored upon exit from runahead mode. The processor starts fetching instructions starting at the address of the instruction that caused entry into runahead mode.

Our policy is to exit from runahead mode when the data of the blocking load returns from memory. An alternative policy is to exit some time earlier using a timer so that a portion of the pipeline-fill penalty or window-fill penalty [1] is eliminated. We found that this alternative performs well for some benchmarks whereas it performs badly for others. Overall, exiting early performs slightly worse. The reason it performs worse for some benchmarks is that some more second-level cache miss prefetch requests are generated if the processor does not exit from runahead mode early.

A more aggressive runahead implementation may dynamically decide when to exit from runahead mode. Some benchmarks benefit from staying in runahead mode even hundreds of cycles after the original L2 miss returns from memory. We are investigating the potential and feasibility of a mechanism that dynamically decides when to exit runahead mode.

5. Simulation methodology

We used a simulator that was built on top of a micro-operation (uop) level IA-32 architectural simulator that executes Long Instruction Traces (LITs). A LIT is not actually a trace, but a checkpoint of the processor state, including memory, that can be used to initialize an execution-based performance simulator. A LIT also includes a list of “LIT injections,” which are system interrupts needed to simulate events like DMA. Since the LIT includes an entire snapshot of memory, we can simulate both user and kernel instructions, as well as wrong-path instructions.

Table 1 shows the benchmark suites we used. We evaluated runahead execution on LITs that gain at least 10% IPC improvement with a perfect L2 cache. In all, there are 80 benchmarks, comprising 147 LITs. Each LIT is 30 million IA-32 instructions long, and carefully selected to be representative of the overall benchmark. Unless otherwise stated, all averages are harmonic averages over all 147 LITs.

Suite	No. of Bench.	Description or Sample Benchmarks
SPEC* CPU95 (S95)	10	vortex + all fp except fppp [26]
SPECfp*2K (FP00)	11	most SPECfp2K [26]
SPECint*2K (INT00)	6	some SPECint2K [26]
Internet (WEB)	18	SPECjbb* [26], WebMark*2001 [4]
Multimedia (MM)	9	MPEG, speech recognition, Quake*
Productivity (PROD)	17	SYSmark*2k [4], Winstone* [28]
Server (SERV)	2	TPC-C*, TimesTen* [27]
Workstation (WS)	7	CAD, Verilog*

Table 1. Simulated Benchmark Suites.

The performance simulator is an execution-driven cycle-accurate simulator that models a superscalar out-of-order execution microarchitecture similar to that of the Intel Pentium 4 processor [13]. The simulator includes a detailed memory subsystem that fully models buses and bus contention. We evaluate runahead for two baselines. The first is a 3-wide machine with microarchitecture parameters similar to the Intel Pentium 4 processor, which we call the “current baseline.” The second is a more aggressive 6-wide machine with a pipeline twice as deep and buffers four times as large as those of the current baseline, which we call the “future baseline.” Table 2 gives the parameters for both baselines. Note that both baselines include a stream-based hardware prefetcher [14]. Unless otherwise noted, all results are relative to a baseline using this prefetcher (HWP).

6. Results

We first evaluate how runahead prefetching performs compared to the stream-based hardware prefetcher. Figure 3 shows the IPC of four different machine models. For each suite, bars from left to right correspond to: 1) a model with no prefetcher and no runahead (current baseline without the prefetcher), 2) a model with the stream-based prefetcher but without runahead (current baseline), 3) a model with runahead but no prefetcher, and 4) a model with the stream-based prefetcher and runahead (current baseline with runahead). Percentage numbers on top of the bars are the IPC improvements of runahead execution (model 4) over the current baseline (model 2).

PARAMETER	CURRENT	FUTURE
Processor Frequency	4 GHz	8 GHz
Fetch/Issue/Retire Width	3	6
Branch Misprediction Penalty	29 stages	58 stages
Instruction window size	128	512
Scheduling window size	16 int, 8 mem, 24 fp	64 int, 32 mem, 96 fp
Load and store buffer sizes	48 load, 32 store	192 load, 128 store
Functional units	3 int, 2 mem, 1 fp	6 int, 4 mem, 2 fp
Branch predictor	1000-entry 32-bit history perceptron [15]	3000-entry 32-bit history perceptron
Hardware Data Prefetcher	Stream-based (16 streams)	Stream-based (16 streams)
Trace Cache	12k-uops, 8-way	64k-uops, 8-way
Memory Disambiguation	Perfect	Perfect

Memory Subsystem

L1 Data Cache	32 KB, 8-way, 64-byte line size	64 KB, 8-way, 64-byte line size
L1 Data Cache Hit Latency	3 cycles	6 cycles
L1 Data Cache Bandwidth	512 GB/s, 2 accesses/cycle	4 TB/s, 4 accesses/cycle
L2 Unified Cache	512 KB, 8-way, 64-byte line size	1 MB, 8-way, 64-byte line size
L2 Unified Cache Hit Latency	16 cycles	32 cycles
L2 Unified Cache Bandwidth	128 GB/s	256 GB/s
Bus Latency	495 processor cycles	1008 processor cycles
Bus Bandwidth	4.25 GB/s	8.5 GB/s
Max Pending Bus Transactions	10	20

Table 2. Parameters for Current and Future Baselines.

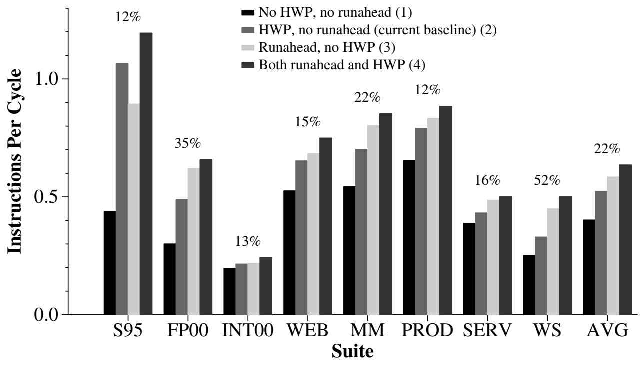

Figure 3. Runahead on current model.

The model with only runahead outperforms the model with only HWP for all benchmark suites except for SPEC95. This means that runahead is a more effective prefetching scheme than the stream-based hardware prefetcher for most of the benchmarks. Overall, the model with only runahead (IPC:0.58) outperforms the model with no HWP or runahead (IPC:0.40) by 45%. It also outperforms the model with only the HWP (IPC:0.52) by 12%. But, the model that has the best performance is the one that leverages both the hardware prefetcher and runahead (IPC:0.64). This model has 58% higher IPC than the model with no HWP or no runahead. It has 22% higher IPC than the current baseline. This IPC improvement over the current baseline ranges from 12% for the SPEC95 suite to 52% for the Workstation suite.

6.1. Interaction between runahead and the hardware data prefetcher

If runahead execution is implemented on a machine with a hardware prefetcher, the prefetcher tables can be trained and new prefetch streams can be created while the processor is in runahead mode. Thus, runahead memory access instructions not only can generate prefetches for the data they need, but also can trigger hardware data prefetches. These triggered hardware data prefetches, if useful, would likely be initiated much earlier than they would be on a machine without runahead. We found that if the prefetcher is accurate, using runahead execution on a machine with a prefetcher usually performs best. However, if the prefetcher is inaccurate, it may degrade the performance improvement of a processor with runahead.

Prefetches generated by runahead instructions are inherently quite accurate, because these instructions are likely on the program path. There are no traces whose performance is degraded due to the use of runahead execution. The IPC improvement of runahead execution ranges from 2% to 401% over a baseline that does not have a prefetcher. This range is from 0% to 158% over a baseline with the stream-based prefetcher.

In this section we show the behavior of some applications that demonstrate different patterns of interaction between runahead execution and hardware data prefetcher. Figure 4 shows the IPCs of a selection of SPECfp2k and SPECint2k benchmarks on the four models. The number on top of each benchmark denotes the percentage IPC improvement of model 4 over model 2. For gcc, mcf, vortex, mgrid, and swim, both runahead execution and the hardware prefetcher alone improve the IPC. When both are combined, the performance of these benchmarks is better than if either technique was used alone.

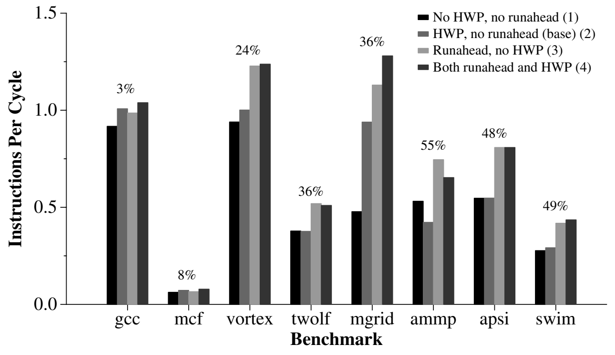

Figure 4. Prefetcher-runahead interaction.

Sometimes, having a prefetcher on a machine that implements runahead execution degrades performance. Twolf, and ammp are examples of this case. For twolf, the prefetcher causes bandwidth contention by sending useless prefetches. For ammp, the prefetcher sends out inaccurate prefetches, which causes the IPC of a machine with runahead and the streaming prefetcher to be 12% less than that of a machine with runahead and no prefetcher.

6.2. Runahead and large instruction windows

This section shows that with the prefetching benefit of runahead execution, a processor can attain the performance of a machine with a larger instruction window.

Figure 5 shows the IPCs of four different models. The leftmost bar of each suite is the IPC of the current baseline with runahead. The other three bars show machines without runahead and with 256, 384, and 512-entry instruction windows, respectively. Sizes of all other buffers of these large-window machines are scaled based on the instruction window size. The percentages on top of the bars show the IPC improvement of the runahead machine over the machine with the 256-entry window.

On average, runahead on the 128-entry window baseline has an IPC 3% greater than a model with a 256-entry window. Also, it has an IPC within 1% of that of a machine with a 384-entry window. For two suites, SPECint2k and Workstation, runahead on the current baseline performs 1–2% better than the model with the 512-entry window. For SPEC95, runahead on the 128-entry window has a 6% lower IPC than the machine with 256-entry window. This is due to the fact that this suite contains mostly floating-point benchmarks that have long latency FP operations. A 256-entry window can tolerate the latency of those operations better

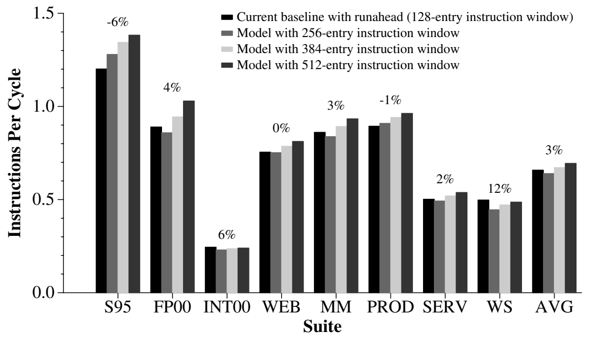

Figure 5. Performance of Runahead versus models with larger instruction windows.

than a 128-entry one. Runahead execution, with its current implementation, does not offer any solution to tolerating the latency of such operations. Note that the behavior of the SPECfp2k suite is different. For SPECfp2k, runahead performs 2% better than the model with the 256-entry window, because traces for this suite are more memory-limited than FP-operation limited.

6.3. Effect of a better frontend

There are two reasons why a machine with a more aggressive frontend would increase the performance benefit of runahead:

1. A machine with a better instruction supply increases the number of instructions executed during runahead, which increases the likelihood of prefetches.
2. A machine with better branch prediction decreases the likelihood of an INV branch being mispredicted during runahead. Therefore, it increases the likelihood of correct-path prefetches. In effect, it moves the "divergence point" later in time during runahead.

If a mispredicted INV branch is encountered during runahead mode, it doesn't necessarily mean that the processor cannot generate useful prefetches that are on the program path. The processor can reach a control-flow independent point, after which it continues on the correct program path again. Although this may be the case, our data shows that it is usually better to eliminate the divergence points. Averaged over all traces, the number of runahead instructions pseudo-retired before the divergence point is 431 per runahead mode entry, and the number of runahead instructions pseudo-retired after the divergence point is 280. The number of L2 miss requests generated before the divergence point is 2.38 per runahead entry, whereas this number is 0.22 after the divergence point. Hence, far fewer memory-to-L2 prefetches per instruction are generated after the divergence point than before.

Figure 6 shows the performance of runahead execution as the frontend of the machine becomes more ideal. Models with "Perfect Trace Cache" model a machine that never misses in the trace cache and whose trace cache is not limited by fetch breaks and supplies the maximum instruction fetch bandwidth possible. In this model, the traces are formed using a real branch predictor.

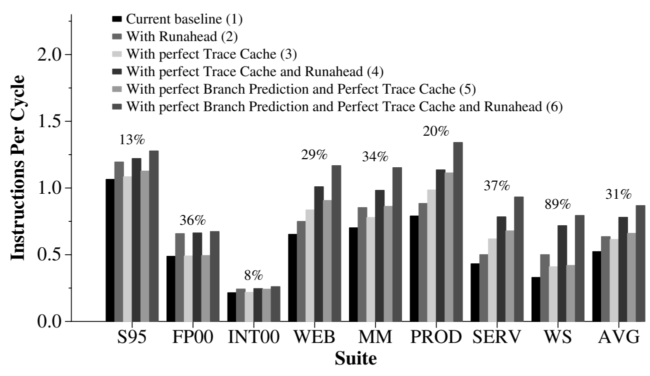

Figure 6. Performance of Runahead as the frontend of the machine gets more ideal.

As the frontend becomes more ideal, the performance improvement of runahead execution increases. For all suites except for SPECint2k, the IPC improvement between bars 5 and 6 (shown as the percentage above the bars) is larger than the IPC improvement between bars 1 and 2. As mentioned before, runahead execution improves the IPC of the current baseline by 22%. Runahead on the current baseline with a perfect trace cache and a real branch predictor improves the IPC of that machine by 27%. Furthermore, runahead on the current baseline with a perfect trace cache and branch prediction improves the performance of that machine by 31%. On a machine with perfect branch prediction and a perfect trace cache, the number of pseudo-retired instructions per runahead entry increases to 909 and the number of useful L2 miss requests generated increases to 3.18.

6.4. Effect of the runahead cache

This section shows the importance of handling store-to-load data communication during runahead execution. We evaluate the performance difference between a model that uses a 512-byte, 4-way set associative runahead cache with 8-byte lines versus a model that does not perform memory data forwarding between pseudo-retired runahead stores and their dependent loads (but still performs data forwarding through the store buffer). In the latter model, instructions dependent on pseudo-retired stores are marked INV. For this model, we also assume that communication of INV bits through memory is handled correctly and magically without any hardware and performance cost, which gives an unfair advantage to that model. Even with this advantage, a machine that does not perform store-load data communication through memory during runahead mode loses much of the performance benefit of runahead execution.

Figure 7 shows the results. In all suites but SPECfp2k, inhibiting store-load data communication through memory significantly decreases the performance gain of runahead execution. The overall performance improvement of runahead without using the runahead cache is 11% versus 22% with the runahead cache. For all suites except SPECfp2k and Workstation, the IPC improvement for the model without the runahead cache remains well under 10%. The improvement ranges from 4% for SPEC95 to 31% for SPECfp2k. The runahead cache used in this study correctly handles 99.88% of communication between pseudo-retired stores and their dependent loads. It may be possible to achieve similar performance using a smaller runahead cache, but we did not tune its parameters.

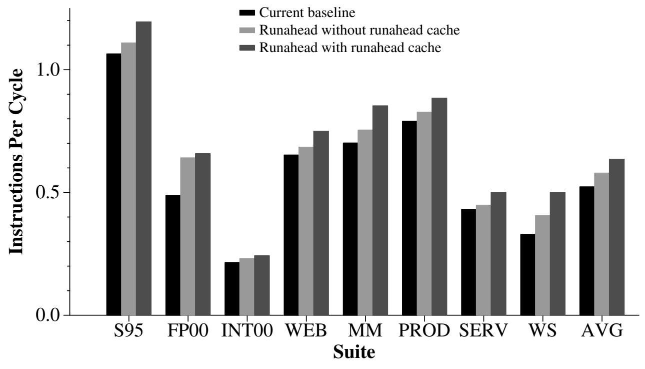

Figure 7. Performance of Runahead with and without the runahead cache.

6.5. Runahead execution on the future model

Figure 8 shows the IPC of the future baseline machine, the future baseline with runahead, and the future baseline with a perfect L2 cache. The number on top of each bar is the percentage IPC improvement due to adding runahead to the future baseline. Due to their long simulation times, the benchmarks art and mcf were excluded from evaluation for the future model. Runahead execution improves the performance of the future baseline by 23%, increasing the average IPC from 0.62 to 0.77. This data shows that runahead is also effective on a wider, deeper, and larger machine.

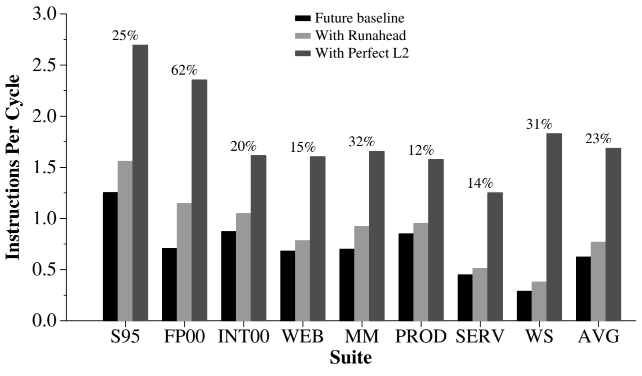

Figure 8. Runahead on the future model.

The effect of runahead on a machine with a better frontend becomes much more pronounced on a larger machine model, as shown in Figure 9. The number on top of each bar is the percentage improvement due to runahead. The average IPC of the future baseline with a perfect instruction supply is 0.90 whereas the IPC of the future baseline with a perfect instruction supply and runahead is 1.38, which is 53% higher. This is due to the fact that branch mispredictions and fetch breaks, which adversely affect the number of useful instructions the processor can pre-execute during runahead mode, are costlier on a wider and larger machine.

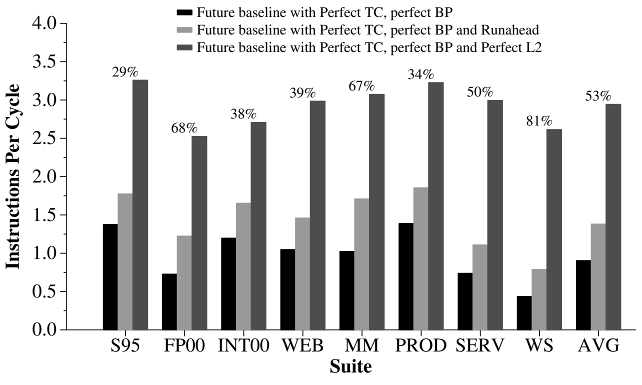

Figure 9. Effect of a perfect frontend on Runahead performance on the future model.

7. Conclusion

Main memory latency is a significant performance limiter of modern processors. Building an instruction window large enough to tolerate the full main memory latency is a very difficult task. Runahead execution achieves the performance of a larger window by preventing the window from stalling on long-latency operations. The instructions executed during runahead mode provide useful prefetching, which improves the IPC performance of an aggressive baseline processor by 22%. This baseline includes a stream-based hardware prefetcher, so this improvement is in addition to the improvement provided by hardware prefetching.

As an extension to the mechanism described in the paper, we are investigating a more aggressive runahead execution engine, which will dynamically extend the length of runahead periods and make use of more of the information exposed by runahead mode. By architecting a mechanism to communicate information between runahead and normal modes, we hope we can unleash more of the potential of runahead execution, allowing us to build even more latency-tolerant processors without having to resort to larger instruction windows.

Acknowledgements

We thank Tom Holman, Hyesoon Kim, Konrad Lai, Srikanth Srinivasan, and the anonymous referees for their help, insights, and comments on this work; and the folks in the Desktop Platforms Group that built and provided us with the simulator for this research. We also thank the other members of the HPS and Intel Labs research groups for the fertile environments they help create. This work was supported by an internship provided by Intel.

References

[1] J. L. Aragón, J. González, A. González, and J. E. Smith. Dual path instruction processing. In *Proceedings of the 2002 International Conference on Supercomputing*, 2002.

[2] J. Baer and T. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, 1991.

[3] R. Balasubramonian, S. Dwarkadas, and D. H. Albonesi. Dynamically allocating processor resources between nearby and distant ILP. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[4] BAPCo*. *BAPCo Benchmarks*. http://www.bapco.com/.

[5] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[6] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, 1999.

[7] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[8] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, 2001.

[9] R. Cooksey. *Content-Sensitive Data Prefetching*. PhD thesis, University of Colorado, Boulder, 2002.

[10] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 1997 International Conference on Supercomputing*, 1997.

[11] J. D. Dundas. *Improving Processor Performance by Dynamically Pre-Processing the Instruction Stream*. PhD thesis, University of Michigan, 1998.

[12] L. Gwennap. Intel's P6 uses decoupled superscalar design. *Microprocessor Report*, Feb. 1995.

[13] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the Pentium 4 processor. *Intel Technical Journal*, Feb. 2001. Q1 2001 Issue.

[14] Intel Corporation. *Intel Pentium 4 Processor Optimization Reference Manual*, 1999.

[15] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *Proceedings of the Seventh IEEE International Symposium on High Performance Computer Architecture*, 2001.

[16] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[17] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990.

[18] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2), 1999.

[19] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, 1981.

[20] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[21] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[22] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[23] A. Moshovos and G. S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, 1997.

[24] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992.

[25] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, 2002.

[26] The Standard Performance Evaluation Corporation. *Welcome to SPEC*. http://www.spec.org/.

[27] TimesTen, Inc. *TimesTen*. http://www.timesten.com/.

[28] VeriTest*. *Ziff Davis Media benchmarks*. http://www.etestinglabs.com/benchmarks/.

[29] M. V. Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions on Electronic Computers*, 14(2), 1965.

[30] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2), Apr. 1996.

[31] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.