Efficient Runahead Execution: A Cost and Complexity-Effective Alternative to Large Instruction Windows to Tolerate Long Memory Latencies

Onur Mutlu
Department of Electrical and Computer Engineering
The University of Texas at Austin
onur@ece.utexas.edu

Abstract

Today’s processors are facing ever-longer main memory latencies in terms of processor cycles. These latencies will continue to increase as the rate of improvement in processor speed continues to exceed the rate of improvement in DRAM memory speed. For almost a decade, high-performance processors have relied on out-of-order execution to tolerate long latencies. Out-of-order execution tolerates latency by buffering instructions in an instruction window and finding independent instructions to execute in that window. Unfortunately, we have already reached the point where the size of an instruction window that can tolerate latencies to DRAM is prohibitively large in terms of design complexity and power consumption. And, the problem keeps getting worse. In this proposal, we describe our solution to the memory latency problem, runahead execution, which achieves the latency tolerance of a large instruction window without having to build it. We show that adding runahead execution to a processor with a 128-entry instruction window and an aggressive streaming hardware data prefetcher improves the IPC (Instructions Per Cycle) performance by 22% on average for 80 benchmarks. We also show that runahead execution on a 128-entry instruction window performs within 1% of a processor with no runahead execution and a 384-entry instruction window. Runahead execution achieves this performance improvement at a very small hardware cost without significantly increasing the processor complexity and utilizing otherwise idle processor cycles. Our research objective is to further increase the efficiency and effectiveness of runahead execution. To this end, we propose to examine novel combined hardware/software techniques that increase the energy-efficiency and performance of runahead execution processors.

1. Problem Description

High-performance processors execute instructions out of program order to tolerate long latencies and extract instruction-level parallelism. However, these processors retire instructions in program order to support precise exceptions [51]. If the execution of a long-latency instruction is not complete, that instruction and instructions following it in the sequential instruction stream cannot be retired. Incoming instructions fill the instruction window if the window is not large enough. Once the window becomes full, the processor cannot place new instructions into the window and stalls. This resulting stall is called a full-window stall. It prevents the processor from finding independent instructions to execute in order to tolerate the long latency. The processor’s latency tolerance can be increased by increasing the size of the instruction window such that no full-window stalls occur when a long-latency instruction blocks retirement. However, this is a challenging task due to the design complexity, verification difficulty, and increased power.
consumption of a large instruction window [12, 21]. In fact, Palacharla et al. showed that the complexity and delay of many hardware structures on the critical path increase quadratically with instruction window size [44].

Unfortunately, today’s main memory latencies are so long that out-of-order processors require very large instruction windows to tolerate them. A cache miss to main memory costs about 128 cycles on an Alpha 21264 [59] and 330 cycles on a Pentium-4-like processor [52]. As processor and system designers continue to push for smaller cycle times and larger memory modules and memory designers continue to push for higher bandwidth and capacity, main memory latencies will continue to increase in terms of processor cycles [60, 59]. Figure 1 shows that a current processor, modeled after the Pentium 4, with a 128-entry instruction window, a 512 KB L2 cache, and an aggressive hardware prefetcher spends 68% of its execution cycles in full-window stalls (Processor 1). If the L2 cache is made perfect (Processor 2), the processor wastes only 30% of its cycles in full-window stalls, indicating that long-latency L2 misses are the single largest cause of full-window stalls in Processor 1. On the other hand, Processor 3, which has a 2048-entry instruction window and a 512 KB L2 cache spends only 33% of its cycles in full-window stalls. Hence, a processor with a large instruction window tolerates the main memory latency much better than a processor with a small instruction window. Figure 1 also shows that 49% IPC improvement is possible with a 2048-entry instruction window, and 120% IPC improvement is possible if all L2 cache misses are eliminated. Therefore, significant potential exists to improve the performance of the state-of-the-art out-of-order processors by improving their tolerance to long main memory latencies.

Our main research objective is to improve the memory latency tolerance of high-performance, out-of-order processors with simple, implementable, and energy-efficient mechanisms without resorting to building complex and power-hungry large instruction windows. The purpose of this proposal is to describe our recent work to achieve this objective and to propose future research topics that we intend to study.

Several proposals have been made to mitigate the effects of long main memory latencies in high performance processors. We briefly review those proposals in Section 2. Our solution to the problem, runahead execution, is described in Section 3. In Section 4, we describe the shortcomings of our solution and propose future research areas that we intend to study to make runahead execution more efficient and effective.
2. Previous Research

Memory access is a very important long-latency operation that has concerned researchers for a long time. This section classifies the relevant approaches into five categories (caching, prefetching, out-of-order processing, enhanced in-order processing, multithreading) and briefly describes the shortcomings of the proposed approaches.

2.1. Related Research in Caching

Caches [58] tolerate memory latency by exploiting the temporal and spatial reference locality of applications. Kroft [30] 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. Caches are very effective and are widely used in modern processors. However, their effectiveness is limited due to two main reasons. First, some applications exhibit poor temporal and spatial locality. Second, cache miss penalties are prohibitively large, so, even with a very low but non-zero cache miss rate, the effective memory hierarchy access time remains high. To reduce the cache miss rates, researchers developed software, hardware, and thread-based prefetching techniques.

2.2. Related Research in Prefetching

Software prefetching techniques [29, 8, 37, 34] are effective for applications where the compiler can statically predict which memory references will cause cache misses. For many integer applications this is not a trivial task. These techniques also insert prefetch instructions into applications, increasing instruction bandwidth requirements.

Hardware prefetching techniques [19, 49, 27, 4, 26, 47, 13] 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. These techniques work well if the reference stream of the running application is regular. Streaming and striding prefetchers that are designed to capture regular patterns are already employed in processors [22, 54]. The design of hardware prefetchers that can capture both irregular and regular reference streams is still an open research topic. Hardware prefetching techniques also degrade performance due to cache pollution and unnecessary bandwidth consumption, if the accuracy of the hardware prefetcher is low.

Thread-based prefetching techniques [11, 33, 62, 48] use idle thread contexts on a multithreaded processor to run threads that help the primary thread [9]. 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. These techniques also require hardware support for executing multiple threads simultaneously.

2.3. Related Research in Out-of-order Processing

Out-of-order execution [56] has long been used to tolerate the latency of memory accesses. Precise exception support was first incorporated into out-of-order execution by [46]. The latency tolerance provided by an out-of-order processor that supports precise exceptions is limited by the instruction window size. Although out-of-order execution has been widely employed to achieve high performance, to tolerate today’s memory latencies, the instruction window of an out-of-order processor needs to be very large, as shown in the previous section. Therefore, several researchers focused on augmenting small instruction windows or building large instruction windows to enhance out-of-order execution’s capability to tolerate long main memory latencies.

Balasubramonian et al. [5] proposed a mechanism to execute future instructions when a long-latency instruction blocks retirement in an out-of-order processor. 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. Also this mechanism adds significant complexity to the register file and scheduler entries, requiring counters that need to be incremented every cycle, and adds complex control structures to manage communication between the future thread and the primary thread.

Lebeck et al. [31] 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.

Pai and Adve [43] proposed a compiler technique that increases the memory latency tolerance of a small instruction window. In their mechanism, the compiler restructures the code such that long-latency misses occur in clusters, which can be captured by a fairly small instruction window. Unfortunately, their proposed algorithm is effective only for benchmarks where the memory reference pattern is regular and predictable at compilation time.

Several recent papers [36, 35, 14] proposed more efficient use of the resources in out-of-order processors so that small instruction windows do not stall as frequently as they do. These proposals allow the instructions to release the load buffer, store buffer, and physical register file entries before the instructions are retired, under certain conditions. An orthogonal scheme that delays the allocation of physical registers until they are needed was also proposed [20]. With these schemes, instructions do not unnecessarily hold on to resources that may be required by other instructions in the instruction window. However, these mechanisms do not allow the instruction window (reorder buffer) entries to be released, which reduces their memory latency tolerance.

Akkary et al. [2] and Cristal et al. [15] proposed using checkpointing [24] to build very large instruction windows. These proposals claim to increase the size of the instruction window such that they can accommodate thousands of instructions by taking checkpoints infrequently. However, they do not show how to build power-efficient store buffers—an integral part of large instruction windows. These previous papers also do not address the complexity and power issues that still exist in their proposals for large instruction windows.

2.4. Related Research in Enhanced In-order Processing

In-order execution is unable to tolerate the latency of cache misses. To increase the memory latency tolerance of in-order execution, two schemes have been proposed.

Runahead processing [16] 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 long-latency data cache and instruction cache misses on an in-order processor that does not employ any hardware prefetching techniques [16, 17]. Unfortunately, runahead processing on an in-order execution machine suffers from the major disadvantage of in-order execution: the inability to tolerate the latency of multi-cycle operations, such as first-level cache misses and floating point operations. For this reason, the performance of an in-order runahead processor is significantly worse than an out-of-order processor. In our research, we aim to improve the performance of the higher-performance out-of-order processors, which are the state-of-the-art in high-performance computing.

Barnes et al. [7] proposed two-pass pipelining to reduce the stalls caused by unpredictable multi-cycle operations
in an in-order processor. In their proposal, the execution pipeline of an in-order processor is replicated. Instructions that miss in the data cache do not stall the first pipeline. Instead, they and their dependents are deferred to the second pipeline. While instructions dependent on cache misses are executed in the second pipeline, independent instructions can continue to be executed in the first pipeline. This design increases the tolerance of in-order processors to cache misses. There are three disadvantages of two-pass pipelining. First, it requires a replicated back-end execution pipeline and auxiliary structures, which increase the complexity of the processor significantly. Second, the performance of this design is limited because there is no way to avoid stalls in the second pipeline even if the instructions that are being processed in the second pipeline are independent of each other. Hence, an out-of-order processor would have higher performance than two-pass in-order execution. Third, the tolerance of the two-pass design to long main memory latencies is limited by the size of structures that are on the critical path. To tolerate long main memory latencies, the complexity of the two-pass pipeline would need to increase significantly.

2.5. Related Research in Multithreading

Multithreading has been used to tolerate long main memory latencies. A multithreaded machine [50, 45, 42, 23, 57] has the ability to process instructions from several different threads without performing context switches. The processor maintains a list of active threads and decides which thread’s instructions to issue into the machine. When one thread is stalled waiting for a cache miss to be serviced, instructions from non-stalled threads can be issued and executed. This improves the overall throughput of the processor.

Multithreading has been shown to improve the overall performance of the processor. However, the performance of each individual thread is not improved by multithreading. In fact, it is likely that the performance of each thread is degraded since resources must be shared between all active threads. Hence, multithreading can tolerate main memory latencies when running a multiprogrammed or multithreaded workload, but provides no benefit on a single-threaded application. Our goal in this research is to improve the memory latency tolerance of single-threaded applications or each individual thread in a multithreaded application.

Hardware schemes to parallelize single-threaded applications have been proposed. In multiscalar processors [18], a program’s instruction stream is divided into tasks which are executed concurrently on several processing units. Processing units are organized as a ring and values are communicated between different tasks using the ring interconnect. Multiscalar processors can tolerate a long-latency cache miss in one task by executing independent operations in other tasks. Unfortunately, multiscalar processors require significant hardware complexity to communicate register and memory values between different tasks. Also, how to break a single sequential instruction stream into parallel tasks is a hard problem for many applications.

Dynamic multithreading [1] was also proposed to parallelize a single-threaded application at run time. A dynamic multithreading processor creates threads automatically at procedure and loop boundaries and executes them speculatively on a simultaneous multithreading pipeline. For example, when a call instruction is reached, a speculative thread is spawned to execute on a separate hardware thread context, starting with the next sequential instruction after the call. The non-speculative parent thread executes the function call whereas the spawned speculative thread does not. A cache miss encountered in the non-speculative thread can be tolerated by executing independent instructions in speculative threads. Dynamic multithreading, however, comes at a significant hardware cost which includes a simultaneous multithreading pipeline, support for spawning speculative threads, support for communicating data values between speculative and non-speculative threads, and support for detecting value mispredictions in speculative threads.
3. Proposal: Runahead execution

In our previous work, we have proposed a simple alternative to large, complex instruction windows to tolerate long main memory latencies [40]. This alternative, runahead execution [16, 40], uses a simple algorithm that increases the tolerance of a processor to long-latency memory operations and provides both instruction and data prefetching benefits. We have shown that this algorithm can be easily implemented in a current out-of-order processor and it increases the IPC performance of an aggressive processor model by 22% [40, 41]. We have compared the performance of runahead execution to that of large instruction windows and shown that a processor with runahead execution performs 3% better than a processor with twice the instruction window size and almost as well as a processor with three times the instruction window size. This section briefly describes the operation, the benefits, the cost, and the performance of runahead execution.

3.1. Operation of runahead execution

The mechanism we propose avoids stalling the processor when a long-latency L2 cache miss blocks retirement, preventing new instructions from being placed into the instruction window. When the processor detects that the oldest instruction is a long-latency cache miss that is being serviced, it checkpoints the architectural register state, the branch history register, and the return address stack, records the program counter of the long-latency instruction, and enters a speculative processing mode called “runahead mode.” The processor removes this long-latency instruction from the instruction window. While in runahead mode, the processor continues to execute instructions without updating the architectural state and without blocking retirement due to long-latency cache misses and instructions dependent on them. The results of long-latency cache misses and their dependents are identified as bogus. Instructions that generate or source bogus results are removed from the instruction window so that they do not prevent independent instructions from being placed into the window. Therefore, runahead mode allows the processor to execute more instructions than the instruction window normally permits. Some of the instructions in runahead mode that are independent of long-latency cache misses miss in the instruction, data, or unified caches. Their miss latencies are overlapped with the latency of the runahead-causing cache miss. When the runahead-causing cache miss completes, the processor exits runahead mode, restores the checkpointed state, and resumes normal instruction fetch and execution starting with the runahead-causing instruction. Once the processor returns to “normal mode,” it is able to make faster progress without stalling because some of the data and instructions needed during normal mode have already been prefetched into the caches during runahead mode.

3.2. Benefits of runahead execution

Runahead execution lets the processor do useful processing instead of stalling for hundreds of cycles while a long-latency data cache miss is being serviced. The processing during runahead execution targets the discovery and initiation of long-latency data and instruction accesses to DRAM, and servicing them in parallel with the runahead-causing miss. Besides prefetching these long-latency accesses, runahead execution prefetched data and instructions between levels of the cache hierarchy, trains the hardware data/instruction prefetchers with future access information, and trains the branch prediction structures. Overall, runahead execution prepares the processor for processing future instructions instead of keeping it stalled while the initial L2 miss is being serviced.
3.3. Cost and complexity of runahead execution

The runahead execution mechanism and its implementation on a current high-performance processor are presented in detail in our previous paper [40]. Here we briefly describe the additional hardware needed to implement runahead execution. This hardware consists of the checkpointed architectural registers, branch history register, and return address stack, a single bit (INV bit) associated with every physical register and store buffer entry, and a small, 512-byte store buffer (runahead cache) used to forward data from stores to loads during runahead mode. These structures are shown in Figure 2. None of these added structures are complex or on the critical path of the processor.

The mechanism used to checkpoint the architectural registers is dependent on the microarchitecture. Microarchitectures that store the architectural state in the physical register file can avoid checkpointing the entire architectural register state by checkpointing only the register map that points to the architectural state. Checkpointing of the return address stack can be accomplished without significant hardware cost as described in [28].

An INV bit identifies the entry it is associated with as bogus, i.e. dependent on a long-latency cache miss. The mechanism used to communicate INV bits between dependent instructions is already present in an out-of-order processor, which communicates data values between dependent instructions.

The runahead store buffer is perhaps the most significant area cost of runahead execution. This buffer is used to forward the data values and INV status of store instructions to dependent load instructions during runahead mode. When a store instruction is removed from the instruction window during runahead mode, it writes its data and INV status into the runahead store buffer so that the data and INV status are available to a younger load instruction that accesses the same memory address. The runahead store buffer is physically organized as a cache structure, but it is very small compared to the L1 data cache (512 bytes vs. 32 KB). Hence, its area cost is not very significant. Our simulations show that this buffer is very latency-tolerant and does not need to be accessed in parallel with the data.
cache. Thus, this buffer is not on the processor’s critical path.

As described, adding runahead execution to an out-of-order processor does not significantly increase processor complexity. Yet, as shown in Section 3.4.2, runahead execution attains the performance achieved with larger instruction windows, which are power hungry, complex, and on the critical path [44]. Hence, runahead execution offers a cost and complexity-effective alternative to large windows.

3.4. Performance of runahead execution

We evaluate the performance improvement of adding runahead execution to an aggressive processor model based on Pentium 4 [22]. We use an execution-driven x86 simulator and 80 memory-intensive benchmarks from a very wide variety of suites: SPEC (SPEC95, FP00, INT00), internet (WEB), multimedia (MM), productivity (PROD), server (SERV), and workstation (WS). The benchmark suites we used are described in Section 1. The baseline processor we model is 3-wide, has a 128-entry instruction window (in terms of micro-operations), 29-stage pipeline, 32 KB, 8-way, 3-cycle L1 data cache, 512 KB, 8-way, 16-cycle unified L2 cache, and a 12K-mop, 8-way trace cache. Main memory latency is 495 cycles in the absence of bank conflicts and queueing delays. The processor makes use of an aggressive streaming hardware data prefetcher [54] and a streaming instruction prefetcher. Bandwidth and contention are modeled at all levels of the memory hierarchy. All IPC numbers are in terms of micro-operations per cycle.

<table>
<thead>
<tr>
<th>Suite</th>
<th>No. of Bench.</th>
<th>Description or Sample Benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPEC CPU95 (S95)</td>
<td>10</td>
<td>vortex + all fp except fppp [53]</td>
</tr>
<tr>
<td>SPECfp2K (FP00)</td>
<td>11</td>
<td>most SPECfp2K [53]</td>
</tr>
<tr>
<td>SPECint2K (INT00)</td>
<td>6</td>
<td>some SPECint2K [53]</td>
</tr>
<tr>
<td>Internet (WEB)</td>
<td>18</td>
<td>specjbb [53], webmark2001 [6]</td>
</tr>
<tr>
<td>Multimedia (MM)</td>
<td>9</td>
<td>mpeg, speech recognition, quake</td>
</tr>
<tr>
<td>Productivity (PROD)</td>
<td>17</td>
<td>sysmark2k [6], winstone [61]</td>
</tr>
<tr>
<td>Server (SERV)</td>
<td>2</td>
<td>IPC-C, timesten [55]</td>
</tr>
<tr>
<td>Workstation (WS)</td>
<td>7</td>
<td>CAD, verilog</td>
</tr>
</tbody>
</table>

Table 1. Simulated benchmark suites

3.4.1. Runahead vs. hardware data 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 (baseline without the prefetcher), 2. A model with stream-based prefetcher, but without runahead (baseline), 3. A model with runahead but no prefetcher, 4. A model with stream-based prefetcher and runahead (baseline with runahead). Percentage numbers on top of the bars is the IPC improvement of runahead execution (model 4) over the baseline (model 2).

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 baseline. This IPC improvement over the baseline ranges from 52% for the Workstation suite to 12% for the SPEC95 suite. We conclude that runahead execution and state-of-the-art hardware prefetching interact positively and both should be employed for highest performance.
3.4.2. Runahead vs. large instruction windows

Figure 4 shows the IPC’s of five different processors for each benchmark suite. From left to right, the first bar shows the IPC of the baseline processor. The next bar shows the IPC of the baseline with runahead execution. The other three bars show the IPC’s of processors without runahead and instead with 256, 384, and 512-entry instruction windows, respectively. Percentage numbers on top of the bars show the IPC improvement of adding runahead execution to the baseline.

On average (shown in the rightmost set of bars), adding runahead execution to the baseline processor improves the IPC by 22%. The baseline processor with runahead execution outperforms the processor with a 256-entry window by 3%. Also, the baseline processor with runahead execution has an IPC within 1% of that of the processor with a 384-entry window. That is, runahead execution on a 128-entry window processor attains almost the same IPC as a processor with three times the instruction window size.

3.4.3. Runahead on future processors

We claim that runahead execution will become more important and effective in future generation microprocessors and support this claim with experiments performed on a future processor model.
As processor and system designers continue to push for smaller cycle times and larger memory modules and memory designers continue to push for higher bandwidth and density, main memory latencies will continue to increase in terms of processor cycles [59]. We believe increased latencies will make runahead execution more effective for future processors. To support this hypothesis, we examined the performance of runahead execution on a future processor model. This model is 6-wide, has a 58-stage pipeline, 512-entry window, 1 MB L2 cache, and 1008-cycle main memory latency. Figure 5 shows the IPC of the future baseline machine, future baseline with runahead, and future baseline with perfect L2 cache. 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 execution is effective on a wider, deeper, and larger machine.

![Figure 5. Performance of runahead on the future model](image)

We also show that the performance improvement of runahead execution is bounded by the effectiveness of the processor’s instruction supply mechanism (branch prediction and instruction fetch units). A processor with a perfect trace cache and a perfect branch predictor benefits more from runahead execution shown in Figure 6. The average IPC of the future baseline with perfect instruction supply is 0.90 whereas the IPC of the future baseline with perfect instruction supply and runahead is 1.38, which is 53% higher. Therefore, as computer architects continue to find innovations to improve branch prediction and instruction fetch units, the performance improvement provided by runahead execution will increase.

4. Proposed Future Work

We propose to study three major topics to improve runahead execution in an effort to make it more effective at tolerating long main memory latencies. These areas are:

1. Increasing the efficiency of runahead execution.
2. Increasing the performance improvement provided by runahead execution.
3. Understanding runahead execution.

This section briefly describes each of these areas and our proposed approaches.
4.1. Increasing the efficiency of runahead execution

One major shortcoming of runahead execution is its inefficiency. Even though runahead execution does not require complex structures, it relies on the execution of instructions multiple times to increase performance. We define the efficiency of runahead execution as:

\[
\text{Efficiency} = \frac{\text{Percent Increase in IPC Performance Due to Runahead}}{\text{Percent Increase in Executed Instructions Due to Runahead}}
\]

Our goal is to research techniques that increase the efficiency of runahead execution without reducing, or while increasing, its IPC performance improvement.

4.1.1. Increasing efficiency by reducing the number of instructions executed

As a runahead execution processor speculatively pre-executes portions of the instruction stream, it executes more instructions than a traditional high-performance processor, resulting in higher dynamic energy consumption. Figure 7 shows that, on average, a runahead processor executes 26.5% more instructions than a traditional out-of-order processor to provide a performance improvement of 22%. Unfortunately, for some benchmarks, a runahead processor executes at least 50% more instructions, while providing little or no performance benefit (e.g. bzip2 and parser). In other words, for some benchmarks, runahead execution significantly increases the dynamic energy consumed by the processor without providing any performance benefit.

To reduce the dynamic energy consumed by a runahead processor it is important to reduce the number of extra instructions executed. However, we would like to reduce the number of instructions executed in a runahead processor without reducing the performance improvement provided by runahead execution. We intend to tackle this problem by first identifying the causes of inefficiency in a runahead processor. After the identification of the causes of inefficiency we plan to use hardware and software techniques to eliminate inefficient runahead periods. We believe that techniques that can increase the efficiency of runahead execution also have the potential to increase the efficiency of other speculative pre-execution mechanisms [9, 11].
4.1.2. Increasing efficiency by increasing the performance improvement through increasing the number of useful misses uncovered during runahead periods

Runahead execution is inefficient because each runahead period executes hundreds of instructions only to uncover a few long-latency misses. Our experimental data shows that, on average, 711 instructions are executed in a runahead period to uncover 2.38 long-latency L2 misses. We intend to study techniques that increase the number of long-latency misses generated during each runahead period.

We have observed that not all the instructions executed in runahead mode contribute to the generation of useful prefetches. For example, floating-point instructions cannot generate memory addresses and hence cannot contribute to the generation of useful prefetches. Not executing the instructions that do not contribute to the generation of L2 misses would free up processor resources for other instructions that contribute to the generation of L2 misses. Hence, the number of useful L2 misses generated in a runahead period would increase, which increases both the performance and efficiency of runahead execution. We intend to examine techniques that would dynamically or using hints from the compiler decide which instructions are useful during a runahead period. One approach to increasing efficiency may be dynamically or statically refining the instruction stream executed during runahead mode such that useless instructions are excluded. The advantage of using a dynamic mechanism is that it can adapt the instruction stream to changing program behavior. The advantage of using a static mechanism is that it does not complicate the hardware and the compiler has a larger scope in optimizing the instruction stream.

4.2. Increasing the performance improvement provided by runahead execution

Performance improvement provided by runahead execution can be improved by eliminating some of the shortcomings. This section describes these shortcomings, which we intend to address in our future research.

4.2.1. Cache pollution caused by runahead memory requests

When a cache miss generated in a runahead period is not useful for the correct execution of the program (because it is generated by a wrong-path instruction), it may evict a useful cache block from the first-level and second-level caches. This may cause performance degradation. Eliminating the pollution caused in this way would increase the performance improvement of runahead execution. We have previously proposed filtering techniques to reduce the pollution caused by such requests [38]. These techniques bring the data prefetched in runahead mode only into the first-level cache, in contrast to bringing it into both the first-
level and the second-level caches. If the prefetched data is needed by normal mode while it resides in the first-level cache, it is written back into the second-level cache. This reduces the L2 pollution caused by memory references in runahead. We intend to research other techniques to reduce both L1 and L2 pollution.

4.2.2. Making use of the instruction results generated during runahead mode

Previous papers on runahead execution took it for granted as a prefetch-only technique [16, 40]. Even though the results of most instructions independent of an L2 miss are correctly computed during runahead mode, previous approaches discarded those results instead of trying to utilize them in normal mode execution. We intend to research techniques to utilize the results generated in runahead mode. These results include branch directions, register values, and memory values. If reusing these results during normal mode improves the performance of a runahead processor, we will research techniques to build mechanisms to facilitate such reuse.

4.2.3. Handling of dependent loads in runahead mode: Combining runahead and address/value prediction

One shortcoming of runahead execution is its inability to calculate the addresses of loads that are dependent on an L2 miss. As such, runahead execution is not beneficial for benchmarks where a dependent chain of loads form the critical path (e.g., such as linked-list traversals, tree traversals) of the program. We intend to research techniques that can allow runahead execution to generate prefetches for loads dependent on L2 misses. Techniques that break dependences by predicting the result values [32] of L2-miss loads will be the first focus of our studies in this area. Techniques used to predict the addresses generated by dependent loads [47] may also be utilized in runahead mode for increased performance.

4.2.4. Wrong-path execution during runahead mode

Another shortcoming of runahead execution is its reliance on branch prediction to process the instruction stream. As branch predictors are not perfect, branch mispredictions occur during runahead mode. If a mispredicted branch is dependent on an L2 miss, the branch cannot be resolved and the processor remains on the wrong path for the rest of the runahead period. Although execution on the wrong path is generally useful [39], it is more beneficial to stay on the correct path during runahead execution. To enable staying on the correct path, we intend to examine mechanisms that recognize that a branch is mispredicted during runahead execution even though the branch is unresolvable and put the processor back on correct path. One example of such a mechanism was described in our previous research [3]. We also intend to explore value prediction mechanisms that enable the resolution of branches that are dependent on L2 misses.

4.3. Understanding runahead execution

An orthogonal research area we intend to work on is the analysis of the benefits of runahead execution. Our preliminary analysis shows that runahead execution increases performance mainly because it increases memory-level parallelism by enabling the servicing of multiple misses in parallel with the runahead-causing L2 miss [41]. This observation was independently verified by other researchers [10, 25].

It has been noted that runahead execution is good at capturing and prefetching misses caused by irregular memory accesses that are very hard to capture using a hardware prefetcher [10]. However, previous research did not examine exactly what kind of misses are prefetched by runahead execution. We would like to enhance our understanding of runahead execution by examining the types and patterns of memory access behavior it captures. By understanding the patterns of memory access behavior that can be captured by runahead, we hope to generate new ideas about and insights into:
1. Increasing the performance provided by runahead execution by incorporating mechanisms that can capture patterns that cannot be captured by the current form of runahead execution.

2. Building hybrid prefetchers that can capture both regular and irregular access patterns.

3. Designing algorithms and programs in such a way that they can benefit (more) from runahead execution.

4. Developing compiler algorithms that would reorganize the code such that runahead execution provides (more) performance benefit.

5. Conclusion

This proposal describes runahead execution, a novel way to tolerate long main memory latencies in out-of-order processors. We propose future research topics to increase the energy-efficiency and performance of runahead execution. We also propose to analyze and understand the benefits provided by runahead execution in an effort to provide new insights into how novel algorithms, programs, and hardware techniques can be designed to increase the benefits of runahead execution. By making runahead execution more efficient and effective, we hope to accomplish our main objective of providing a cost and complexity-effective way to tolerate long main memory latencies without resorting to building complex and power-hungry large instruction windows.

6. References


7. Coursework

<table>
<thead>
<tr>
<th>Type</th>
<th>Semester</th>
<th>Unique</th>
<th>Course No.</th>
<th>Course Title</th>
<th>Professor</th>
<th>Letter Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major</td>
<td>Fall 2000</td>
<td>15405</td>
<td>EE 380L</td>
<td>Engineering Programming Languages</td>
<td>Chase</td>
<td>A</td>
</tr>
<tr>
<td>Major</td>
<td>Fall 2000</td>
<td>15490</td>
<td>EE 382N</td>
<td>Communication Networks</td>
<td>Nettles</td>
<td>A</td>
</tr>
<tr>
<td>Major</td>
<td>Fall 2000</td>
<td>15495</td>
<td>EE 382N</td>
<td>Distributed Systems</td>
<td>Garg</td>
<td>A</td>
</tr>
<tr>
<td>Major</td>
<td>Spr 2001</td>
<td>14915</td>
<td>EE 382C</td>
<td>Dependable Computing</td>
<td>Touba</td>
<td>A</td>
</tr>
<tr>
<td>Major</td>
<td>Fall 2001</td>
<td>15652</td>
<td>EE 382N</td>
<td>Superscalar Microprocessor Architecture</td>
<td>John</td>
<td>A</td>
</tr>
<tr>
<td>Major</td>
<td>Spr 2002</td>
<td>15100</td>
<td>EE 382M</td>
<td>VLSI Testing</td>
<td>Touba</td>
<td>A</td>
</tr>
<tr>
<td>Major</td>
<td>Spr 2002</td>
<td>15125</td>
<td>EE 382N</td>
<td>Microarchitecture</td>
<td>Patt</td>
<td>A</td>
</tr>
<tr>
<td>Supporting</td>
<td>Spr 2001</td>
<td>51065</td>
<td>CS 395T</td>
<td>Compilers</td>
<td>Lin</td>
<td>A</td>
</tr>
<tr>
<td>Supporting</td>
<td>Spr 2001</td>
<td>55390</td>
<td>M 394C</td>
<td>Bayesian Statistics</td>
<td>Jefferys</td>
<td>A</td>
</tr>
<tr>
<td>Supporting</td>
<td>Fall 2001</td>
<td>51835</td>
<td>CS 384M</td>
<td>Multimedia Systems</td>
<td>Vin</td>
<td>A</td>
</tr>
<tr>
<td>Other</td>
<td>Fall 2001</td>
<td>15485</td>
<td>EE 371R</td>
<td>Digital Image and Video Processing</td>
<td>Bovik</td>
<td>A</td>
</tr>
</tbody>
</table>