Re-examining Instruction Reuse in Pre-execution Approaches

Sonya R. Wolff Ronald D. Barnes
School of Electrical and Computer Engineering
The University of Oklahoma
Norman, OK 73019
Email: *srwolff@ou.edu*, *ron@ou.edu*

***Abstract*—Long-latency cache accesses cause significant, performance impacting delays for both in-order and out-of-order processors. One method proposed to tolerate these long latency accesses, *runahead pre-execution*, has been shown to produce speedups for both models of execution through increased overlap of data memory accesses. However, the reuse of pre-execution results in an out-of-order runahead processor has previously been shown to provide little additional benefit compared to a simpler prefetching-only pre-execution model. This paper examines runahead pre-execution and the reuse of results for both in-order and out-of-order pre-execution models. As previous research has shown, result reuse on an out-of-order runahead model provides minimal speedups when compared to simple out-of-order runahead. For a selection of SPEC CPU2006 benchmarks, the reproduced results show average speedups of 1.03X and a best case speedup of 1.12X. However, these results do not provide a valid picture of the benefits of result reuse for in-order runahead processors. The fundamentally different behavior of in-order and out-of-order processors greatly affects the performance impact of pre-executionuted instruction reuse. In fact, the addition of reuse to the in-order runahead model results in average speedups of 1.09X and a 1.47X speedup in the best case.**

I. Introduction

As the gap in the relative performance of processors and memories continues to grow, the amount of time computation is stalled waiting on memory has become one of the largest performance hindering conditions. This problem has long been recognized as the "memory wall" [1][2], and there is no indication that this memory gap will decrease significantly in the near future. Several approaches have attempted to address the performance impact of these high memory latencies. Some, such as the introduction of caches, reduce the observed latency of memory. Other strategies attempt to tolerate the latency of slower secondary or tertiary caches, or even main memory itself, by overlapping a long latency memory access with computation or other memory operations. Non-blocking caches [3] allow data accesses to continue even in the presence of a waiting cache miss. Hardware prefetching [4] builds on this strategy by attempting to bring data into some level of the cache before it is requested by an executing instruction. In this way, program execution overlaps long latency cache accesses.

Tolerance of the long, variable latency of cache-missing memory operations is recognized as one of the primary benefits of out-of-order execution [5] and continues to provide its biggest performance benefit over in-order execution [6]. Compilers for in-order architectures use analysis, transformations and scheduling to tolerate anticipated latencies but cannot effectively deal with the unpredictable latency of cache-missing loads [7]. To address such long-latency operations, *runahead pre-execution* was proposed as an execution strategy to increase latency tolerance for both execution models.

During a long-latency memory operation, *runahead pre-execution* provides performance benefits by executing ahead in program code beyond the normal limitations of the architecture, be it in-order [8] or out-of-order [9]. By doing so, memory blocks accessed during runahead are pre-loaded into the cache resulting in a reduction in the long-latency operations during normal execution. An obvious question that arises when contemplating pre-execution is the reuse of instructions that are correctly executed during the runahead process. The conventional runahead approach retains only the aftereffects of prefetching from the pre-execution mode. However, other pre-execution strategies, detailed in Section II, retain the known results of instructions processed during that time.

Runahead reuse might potentially increase efficiency and improve performance by reusing pre-execution results instead of re-executing instructions during the normal execution mode. However, this improvement is not clear and the storage of pre-execution results may require greater complexity and power consumption than re-execution alone. While several approaches have been proposed reusing the results of pre-executed instructions, one study [10] found that out-of-order runahead reuse provided little or no significant speedup. We have reproduce these results, showing that within a similar out-of-order execution model these observations are accurate and result reuse benefits are relatively insignificant. However, on further analysis, the benefit of instruction reuse for an in-order process is a more complex question. On average, the benefit from result reuse for in-order execution is notably higher than in our idealized out-of-order execution evaluations. For some benchmarks, the reuse of results for in-order pipelines provides large speedups. This paper illustrates and examines the reuse strategy within these two different execution models and provide detailed explanations regarding the observed differences between reuse benefits for the two types of processors. An analysis of the reuse behavioral variations for in-order pipelines when executed on various benchmarks is also provided.

II. Pre-execution for Memory- and Instruction-level Parallelism

The impact of memory accesses on performance depends upon how frequently those accesses occur during program execution and where required memory blocks reside in the cache/memory hierarchy. A characterization of SPEC CPU2000 and CPU2006 benchmarks [11] shows that, for all but one benchmark in these suites, load instructions represent at least 30% of the instruction profile and stores account for an additional 10% of the instructions. This means that approximately 40-50% of the program instructions involve a data cache access of some type. If all these cache accesses were L1 cache hits, this instruction distribution would not present a problem for processor pipelines. However, as also seen in [11], for the Core 2 Duo processor, over half of the benchmarks encounter at least 10 L1 and at least one L2 cache miss for every 1000 instructions. Nine of the SPEC benchmarks have more that 5 L2 misses for every 1000 instructions. Most of these misses are the results of load operations. So, at least part of the time, a set of instructions will be waiting many, if not hundreds, of cycles for some memory data to become available. These pipeline stalls increase the number of cycles required to execute a program while, theoretically, not being necessary for obeying data-flow execution. Therefore, it is highly desirable to decrease the impact these stalls have on a program. Of course, it is also desirable to achieve this with a minimum investment in system resources and impact on overall power consumption. Pre-execution is a set of techniques that have been proposed to minimize the impact of long latency cache access with inexpensive modifications to processor hardware. This section discusses several proposed pre-execution systems thereby exploring pre-execution as a viable solution to the problem posed by long latency memory access.

Before defining pre-execution, it is useful to consider the implications of in-order models of execution. Figures 1 and 2 demonstrate the performance hampering impact of strict in-order execution. Figure 1 details an example code fragment that contains two load instructions. The graph in Figure 2 shows the dependences between the instructions in the code fragment. Assuming that in some in-order execution of this example, the first load instruction misses in the data cache, then the subsequent add will stall waiting on the data from that load instruction. Since this consuming add instruction is dependent on the load, the add instruction would be delayed in any data flow obeying execution model. However, the next three instructions are independent of both the load and the add instructions. In in-order execution, instructions that follow the add will also stall even when the subsequent instructions are independent, as in this case. This implicit program-order dependence between all instructions limits the achievable degree of instruction-level parallelism and prevents, in this example, the overlap of the memory accesses for the two load instructions. This decreased instruction-level parallelism will have an especially adverse performance impact if the second load instruction also misses in the cache. It can be argued that overlapping memory accesses is more important than the degree of instruction-level concurrency [12].

```
ld  r1, [r2]
add r3, r1, r4
shl r5, r3, 2
ld  r6, [r5]
mul r7, r6, r8
```

Fig. 1. Code example to demonstrate in-order behavior.

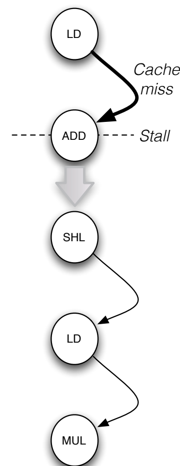

Fig. 2. In-order stall on the result of cache-missing load.

Out-of-order execution is one of the most common microarchitecture strategies in modern general-purpose, high-performance microprocessors [13][14]. Under this model, the processor itself is allowed to determine how to effectively order instruction execution. It attempts to find instruction-grained parallelism by separating program execution from program ordering. Instructions are selected for execution once their instruction dependencies have been resolved. In this way, processor hardware selects independent instructions to execute simultaneously regardless of program order. This execution model allows instruction issue to continue even when a long-latency cache access occurs. In the example shown in Figures 1 and 2, out-of-order execution enables the independent shift, load and multiply instructions to execute during the handling of the first load's cache miss. Also, where the second cache miss occurs, the two long-latency memory accesses can be overlapped in a way that was impossible under the in-order execution model.

In one sense, out-of-order processors are the most common implementation of pre-execution. They hide the long latency resulting from data-cache-missing instructions (in particular data-cache-missing loads) through the fine-grained selection of ready, independent instructions. Out-of-order execution is very general in that the order of instructions selected is limited only by the flow of data, however, this execution model restricts reordering to a markedly limited instruction window.

The original *runahead pre-execution* approach proposed by Dundas and Mudge [8] increased the overlap of memory load operations for a narrow-issue in-order processor. In this model of execution, cache-missing, long-latency loads initiate a pre-execution mode. We refer to this mode as *in-order runahead mode* to distinguish it from other manners of runahead execution. During in-order runahead mode, consumer dependences

(such as the add's dependence on load in Figure 2) do not result in pipeline stalls. Instead, forward execution progress is allowed by deferring the execution of the dependent instruction. Fortunately, many instructions following such cache misses are independent [15]. Execution continues pre-computing the results of independent instructions without retiring the effects of these pre-executed instructions to architectural state. Once the data from the cache miss has been fetched, the pre-executed instructions are flushed and normal execution is restarted starting immediately after the original cache-missing load. While this means that pre-executed instructions are executed again regardless of dependency on the load data, this process enabled blocks loaded during runahead mode to be preloaded into the cache. When the load instructions are re-executed in normal mode, the required block should be fully or partially loaded into the cache. If this approach is successful, loading these blocks will decrease the likelihood of another long-latency memory access since the access's latency should have already been handled during the runahead mode of operation. Dundas showed that this runahead execution improved the overall performance of the program with little additional hardware overhead.

While Dundas' work targeted in-order pipelines, Mutlu extended this approach to out-of-order processors [9]. Mutlu's approach improves the overall performance of an out-of-order processor and provides a viable alternative to very large instruction window sizes. In this technique, *out-of-order runahead mode* starts when a long-latency cache-missing load becomes, by program order, the earliest instruction in the reorder buffer (ROB). The instruction is released from the ROB without committing its result. This frees up dependent instructions which are deferred in a similar manner as [8]. Subsequent independent instructions can then execute which cause the same cache pre-loading effects as in-order pre-execution. When the cache miss that started runahead mode is finished, normal execution returns to the original load instruction. Since the hardware overhead for runahead is significantly less than what is required to increase the instruction window, out-of-order runahead execution provides a power-efficient process for tolerating very long memory latency [16].

Several different runahead approaches have been proposed that preserve not just the cache pre-warming effects of pre-execution but the results of correctly pre-executed instructions as well. Continual flow pipelines could be considered to be one such out-of-order processor model [17]. In this approach, instructions that are dependent on a cache missing load are not held in reservation station entries, but rather flow through execution stages only to be re-scheduled and re-executed out of the reorder buffer once the cache miss has (hopefully) been handled. Very long instruction windows are achieved virtually through the use of check-pointing, and instructions that execute correctly are never re-executed.

In-order models of pre-execution with result reuse include the two-pass pipelining approach [18] in which instructions that are deferred in an "advanced" pre-execution pipeline are executed on a second architectural pipeline. Results from the advanced pipeline are queued for merging into the architectural state by the second pipeline without re-executing the instructions themselves. A related approach, dual-core execution [19] utilizes two cores in a dual-core processor to achieve the benefits of pre-execution with result reuse. A compiler approach, decoupled software pipelining [20], achieves similar behavior (and similar memory latency tolerance) through the static partitioning of loops into different stages based on recurrences through these loops. These different stages are executed on different processor cores, with execution in the first stage scheduled such that non-recurrence, long-latency memory accesses do not stall execution on that stage. Within a single processor pipeline, multi-pass pipelining [21] and in-order continual flow pipelining [22] both provide mechanisms for execution beyond the consumers of long-latency cache-missing loads and retain the valid results pre-executed under this mode. Similarly, the Rock processor [23] developed by Sun, features an *execute ahead* capability which utilizes support for simultaneous multithreading to perform pre-execution with result reuse in an otherwise in-order processor. It is notable that all previous evaluations of each of these in-order approaches show some non-negligible performance benefits from the reuse of such results in apparent conflict with [10].

III. Methodology

The evaluations discussed in Section IV were performed using Soonergy, a cycle accurate architectural and microarchitectural simulator. The simulator models variable stage length in-order and out-of-order pipelines with typical microarchitectural components and a realistic memory hierarchy. The specifications used for our baseline evaluations are detailed in Table I and have been selected in an attempt to replicate, as much as possible, the simulation parameters specified by [10]. Note that these specifications represent an admittedly over-aggressive processor configuration. It should also be noted that for this paper, a conscious decision was made not to use the aggressive stride prefetcher [24] used in [10]. This decision was made because the primary focus of this paper is a detailed examination of the differences between runahead pre-execution on in-order and out-of-order processors only. Experiments including a prefetcher have been performed with similar results as [10] and therefore have been left out of our analysis in this work.

As with any attempt to replicate simulated processor results, exactly matching the original result values produced by different simulator proves to be impossible. Section IV demonstrates a reproduction of similar results and identical trends as seen in [10]. Significant effort was devoted in attempt to reproduce the simulated processor environments used in this work. However, there are inevitable simulated differences, caused by slight variations in the design and behavioral implementation of various structures. Also, the amount of detail any previous work can provide about the operation of their simulator and their simulated architecture is limited.

Beyond variations in the simulator design, differences in observed results are also introduced by the very benchmarks

TABLE I
BASELINE PROCESSOR CONFIGURATIONS

Pipeline A	In-order	8-wide in-order processor pipeline
Pipeline B	Out-of-order	8-wide instruction fetch and issue, 128-entry ROB, 128-entry RS, 256 physical registers, 64-entry LSQ
Common Setup	Execution Core	8 ALUs (with latency): integer ALU (1) integer multiply (8), floating point ALU (4), FP divide (16)
	All Caches	128 entry MSHRs, LRU replacement, 64B line size
	L1 I-Cache	64KB, 4 way, 2 cycle, 128 entry MSHRs 4 ld per cycle
	L1 D-Cache	64KB, 4 way, 2 cycle, 128 entry MSHRs, Write-Through, 4 ld/st per cycle
	Unified L2 Cache	1MB, 32 way, 10 cycle, 128 entry MSHRs, Write-Back, 1 ld/st per cycle
	Memory	Minimum 500 latency, 32 banks, 32B wide, split transactions core-to-memory at 4:1 freq. ratio
	Branch Predictor	64K-entry gshare/PAs hybrid branch predictor with a minimum 20-cycle miss predict penalty

examined. Newer benchmark suites have become available since many of the previous studies were performed, and we have utilized these newer and hopefully more interesting workloads for most of our data analysis. We have compared experiments using the older and newer benchmark suites which produced results that imply similar conclusions. In this work, we have used the newer benchmarks suites for the experiments presented in Section IV.

The binary executables used for experiments are different than those in other works because of compilation differences and the limited program runs. Compliers are different from one platform to another, and for our simulations, the benchmarks examined in this work were complied for 64-bit x86 execution on the Microsoft Windows 7 operating system. Also, as is common, simulations in both the previous work and this paper were performed using only a subsection of the program. In the previous literature, precise information about exactly which benchmark sections were simulated is not typically reported. For the purpose of this paper, the simulations were performed for 250 million instructions where the starting point was chosen from a statistically relevant section of the program [25].

In spite of our best effort, the simulator used for evaluations in this study does not produce identical results to previous simulators used for pre-execution studies. These slight result variations can be seen when comparing results in Section IV to those in [10]. These variations can be attributed to the above mentioned experimental variations; however, the overall behavioral trends of our experiments and previous research is similar.

IV. RESULTS

Simulations were run for the set of C language SPEC CPU2006 benchmarks that were compatible with Microsoft Visual Studio 2010 on Microsoft Windows 7. The benchmark values shown are the weighted average for all reference input where the weight is based upon the total number of instruction executed during a complete program run for each reference input. All programs were executed for 250 million instructions with an additional 25 million instructions used to warm-up hardware structures such as the branch predictor and caches. The results shown in this section exclude this warm-up period.

A. Evaluating pre-execution and reuse of pre-executed results

Six different execution models are examined: in-order (IO), in-order runahead (IORA), in-order runahead with reuse (IO-RARU), out-of-order (OO), out-of-order runahead (OORA) and out-of-order runahead with reuse (OORARU). Figure 3 plots each benchmarks normalized number of executed cycles for each model relative to the number of cycles executed for that benchmark in the in-order model. In the reuse models, instructions that are independent of long latency cache misses and are correctly executed during runahead mode are not re-executed but rather treated as if they could be skipped once normal execution is resumed. In the runahead reuse models, the capacity for reuse is limited only by the ability of the processor to predict branches in order to fetch instructions into the processor pipeline. This is especially true for branches that occur after a cache miss and are on its dependent chain. This idealized reuse model was chosen to match the reuse evaluation in [10].

Since the results in Figure 3 are based on the total number of execution cycles, the shorter the column, the fewer number of cycles necessary to complete program execution. The first and most apparent result is that out-of-order execution always provides a significant speedup in comparison to in-order execution. This is a well observed characteristic. However, it can also be seen that for *omnetpp* and *soplex*, in-order runahead with reuse achieves better performance than the normal out-of-order processor. Also for *gcc, lbm* and *mcf*, the in-order runahead reuse model achieves performance results that are closer to the performance of out-of-order than that of the in-order model.

The out-of-order results shown in Figure 3 largely agree with the results in [10]. Significant speedups are seen for the application of runahead execution, but only a marginal 1.03 X improvement is seen even with the relatively idealized reuse model being evaluated. The speedup from out-of-order runahead reuse is slightly larger than that seen in [10] but still provide little motivation for the complexity involved in preserving the results of runahead reuse. However, the reuse approach provides greater performance improvements for in-order runahead model than the out-of-order model for *astar, bzip, dealII, gcc, h264, lbm, omnetpp*, and *soplex*. The average speedup from reuse in the in-order case is 1.09 X. Notably, the most significant gain from pre-execution reuse in the the out-of-order model is 1.12 X in *mcf* while, in the in-order runahead with reuse model, a 1.48 X speedup is seen from reuse in *lbm*. These differences are examined more closely in the remainder of this section. Note that *gobmk, hmmer, namd,* and *povray* appear to reap little benefit from pre-execution for either in-order or out-of-order configurations regardless of reuse. This is examined more closely in Section IV-B

Figure 4 plots the number of times a benchmark entered runahead mode in increments of 1000 for both in- and out-of-order execution models. Because *mcf* enters runahead significantly more times than any of the other benchmarks, the number of runahead entries is noted beside *mcf*'s columns

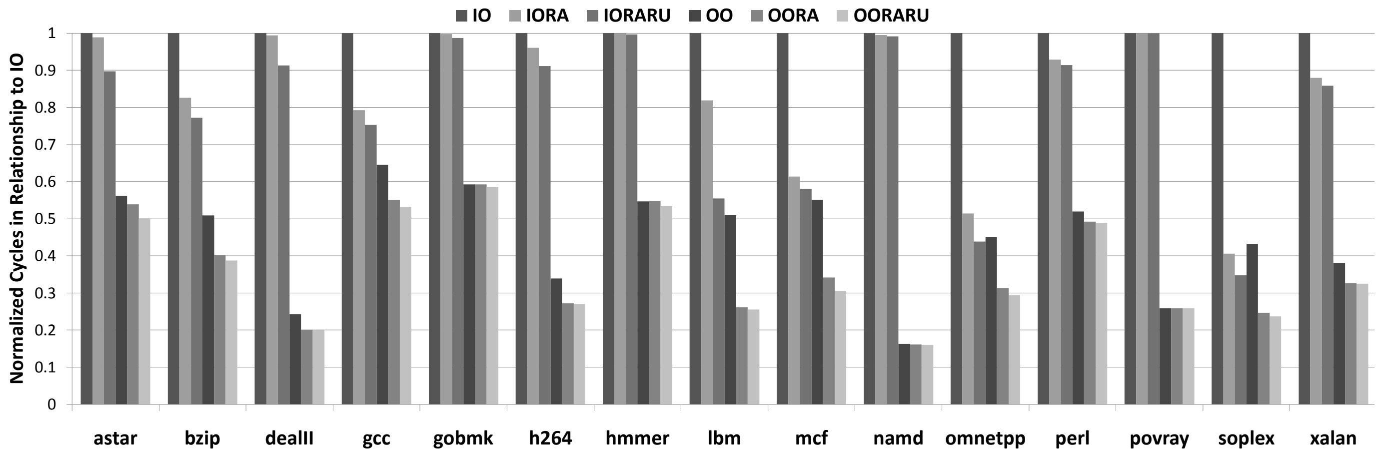

Fig. 3. Normalized cycles executed (normalized to in-order processor)

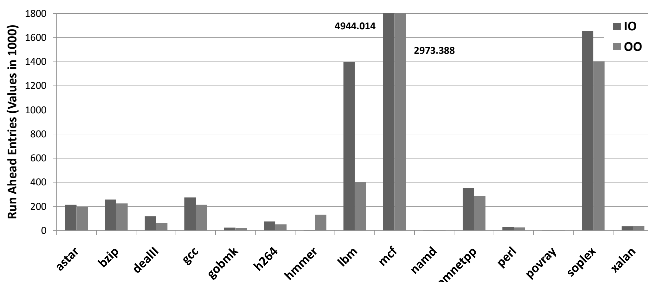

Fig. 4. Number of entries into runahead execution model for in-order and out-of-order models

instead of plotting them directly. As shown in this figure, the benchmarks with little to no improvements from runahead execution seldom enter runahead mode. Runahead mode is entered more often for in-order pipelines than for out-of-order pipelines for all benchmarks except *hmmer*. Since out-of-order runahead execution is only initiated when a long-latency cache miss becomes the oldest instruction in-program-order in the ROB, many L2 cache misses may never reach this point and never initiate a runahead mode. However, almost every L2 data cache miss will result in an in-order stall and initiate runahead execution in the in-order runahead model.

Examining the runahead entires for *hmmer* compared to other benchmarks, it was found that *hmmer* has more L2 misses for out-of-order than for in-order. This behavior is unique to *hmmer* for this benchmark suite and can occur when out-of-order causes thrashing in the cache because load operations that occur later in the serial program kick out cache blocks that earlier, not-yet-issued memory operations will require. As noted in Section IV-B, *hmmer*, even in the out-of-order execution model has few long-latency cache misses. Because of this behavior, it is not surprising that *hmmer* also benefits very little from runahead execution in the out-of-order model.

A runahead reuse pipeline's performance improvement depends on providing to the pipeline valid, reusable instruction results for those instructions that were completely executed during runahead mode. This performance improvement largely depends on how many of these "cleanly" executed instructions occur during a program's run. The number of these instructions is contingent on how many instructions depend on the original runahead-causing load and on the frequency that subsequent memory access can be "cleanly" executed during runahead. A cleanly executed memory access is one in which the address for the memory operation can be calculated (i.e. does not depend on a deferred runahead instruction) and if the access returns the requested information in a timely manner. For the purposes of our evaluations, the acceptable latency represents the L2 access time for pre-execution reuse models and the L1 access time for pre-execution models without result reuse. For a pipeline with reuse, waiting for a cache access to L2 will provide more instructions that can be reused in normal operation mode. When the pipeline is not going to reuse the instruction, the memory access itself is sufficient to cause the cache to be preloaded. Moreover, waiting a significant amount of time for the data to return does not provide as much benefit as issuing as many memory operations as possible during runahead mode. This slight difference in runahead behavior results in improved performance for in-order runahead with reuse. It has no noticeable impact on the performance of out-of-order runahead with reuse.

Figure 5 shows the percentage of the memory instructions that executed cleanly during pre-execution for runahead with reuse models. In the in-order model, all of the in-order benchmarks encounter a greater number of "clean" memory operations than the out-of-order processor. This graph helps explain why, despite the large number of entries into runahead mode, *mcf* has significant less speedup from reuse as the other two high runahead usage programs, *lbm* and *soplex*. In *mcf*, a large number of pre-executed memory accesses miss in the L2 data cache, leaving only around 50% of memory operations to execute cleanly. However, *lbm* and *soplex* have over 80% "clean" memory accesses during pre-execution. The high number of "dirty" memory accesses in *mcf* means that

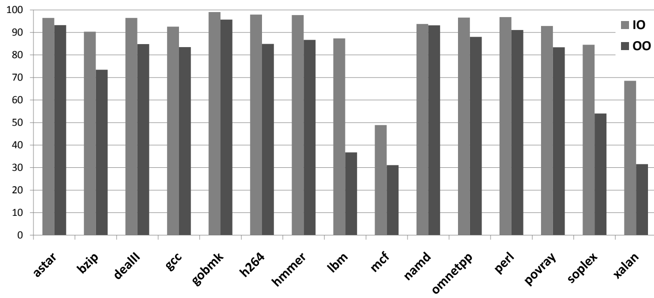

Fig. 5. Percent memory accesses that are "clean" during runahead mode (for runahead with reuse models)

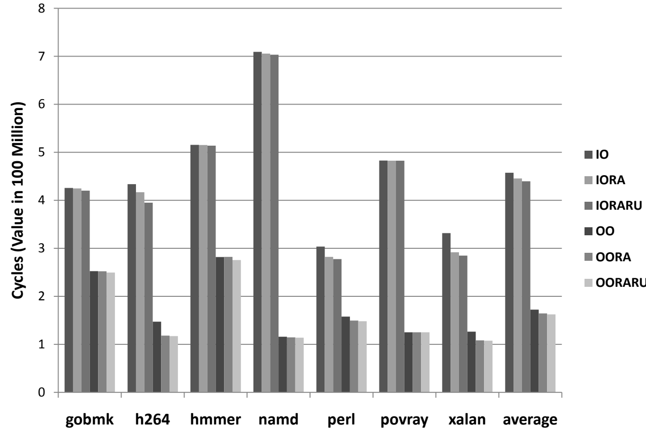

Fig. 6. Execution cycles (low runahead entry)

a great deal of main memory overlap is achieved, but only a small number of independent instructions are left executing correctly and therefore available for reuse.

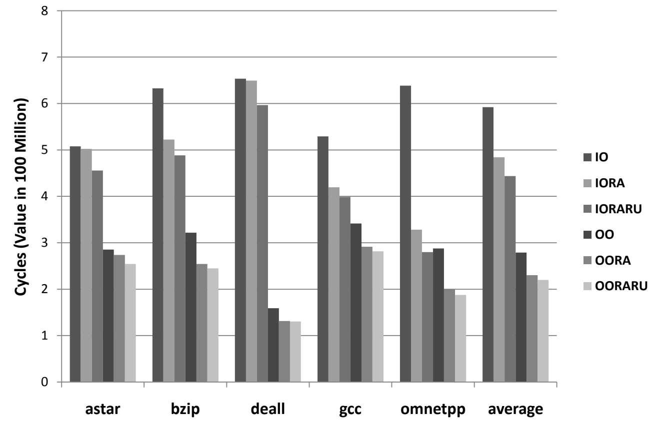

Fig. 7. Execution cycles (medium runahead entry)

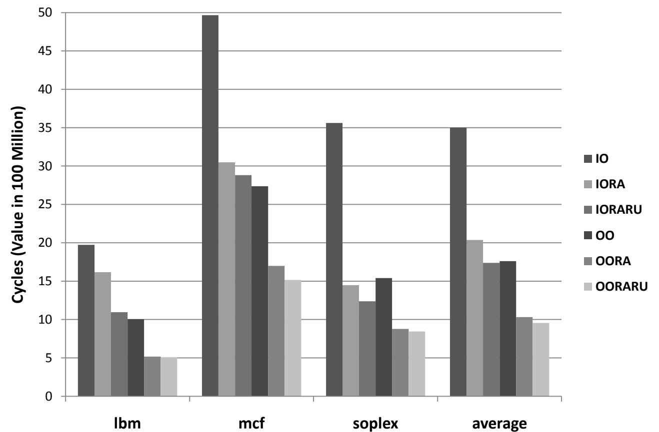

Fig. 8. Execution cycles (high runahead entry)

B. Categorizing application runahead behavior

The previous section dealt with all of the evaluated benchmarks as a unit, but as can be seen from Figure 4, there are really different classes of runahead behavior based upon the number of times the system actually enters the runahead pre-execution mode. We classify all the benchmarks into three groups composed of a high runahead entry set (*lbm, mcf* and *soplex*), a medium runahead entry set (*astar, bzip, deall, gcc,* and *omnetpp*) and a low runahead entry set (*gobmk, h264, hmmer, namd, perl, povray,* and *xalan*). In breaking the entire collection of benchmarks down into these three groups, it becomes clearer how runahead pre-execution behaves in its various models.

The total execution time for the 250 million instruction simulation for the benchmarks grouped in the low runahead entry category are presented in Figure 6 along with the average execution time for these seven benchmarks. Five of these seven benchmarks (*gobmk, h264, perl, povray,* and *xalan*) are the five benchmarks with the fewest number of executed cycles for all of the benchmarks evaluated. *Hmmer* and *namd* have execution times which are similar to most of the benchmarks in the medium runahead entry set shown in Figure 7. However, these two benchmarks have significantly fewer L2 cache misses than any of the benchmarks in the medium set. Figure 9 charts the L2 misses for both low and medium runahead entry sets. Along with *povray*, *hmmer* and *namd* have the lowest number of overall L2 cache misses. This results in a relatively small speedup from runahead pre-execution and a very modest improvement from the addition of result reuse.

The medium runahead entry benchmarks have similar execution times and a higher number of L2 cache misses with the exception of *dealII*. It can also be observed that reuse provides more of a benefit to this set of benchmarks than it does for the low runahead entry set. Notably, *gcc*'s execution time for in-order runahead with reuse begins to approach that of the baseline out-of-order model, and in-order runahead with reuse actually outperforms the out-of-order model for *omnetpp*.

The high runahead entry group (*lbm, mcf* and *soplex*) have execution times that are significantly higher than any benchmark in the other two sets. Because of these extremely high cycle times, the impact of reuse is significantly more for high usage than medium usage even though the drop in the executed cycles with the addition of reuse in Figure 8 appears to be the similar to the decrease in Figure 7. The magnitude,

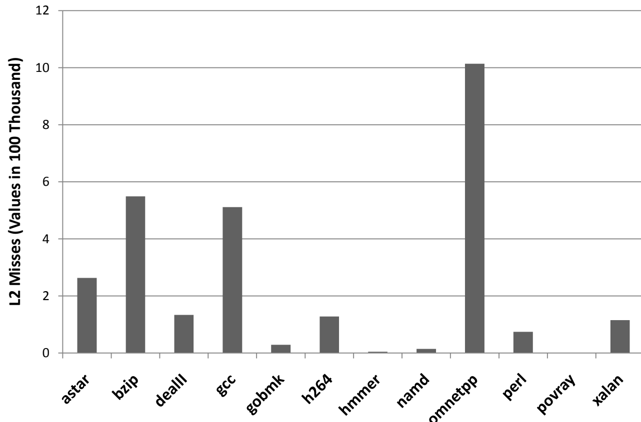

Fig. 9. L2 cache misses (baseline in-order)

TABLE II
REUSE SPEEDUPS

Program Group	In-Order	Out-of-Order
High Runahead Entry	1.172 X	1.080 X
Medium Runahead Entry	1.092 X	1.047 X
Low Runahead Entry	1.013 X	1.011 X

in number of cycles, represented by the decrease in the high usage group reuse cycle time is over five times that of the medium group's decrease in cycle time. As was seen with the medium runahead entry group, *lbm* and *mcf* have execution times approaching that of the baseline out-of-order execution model and *soplex* outperforms it. As was stated earlier, *mcf*'s improvement from reuse is less than that of either *soplex* or *lbm* because of its relatively smaller number of clean memory operations as shown in Figure 5.

Breaking the benchmarks up into three groups makes it easier to observe the variation in behavior between applications. Table II summarizes the speedups from the exploitation of reuse in the two runahead execution models (in-order and out-of-order). The speedups for reuse in the in-order model is higher for all three categories than the speedup for the corresponding out-of-order model; however, even in the high runahead entry group, reuse in the out-of-order model does not achieve the speedup that is seen in the in-order model. This largely confirms the pessimistic results for out-of-order runahead with result reuse seen in [10], while providing some indication that the result reuse merits further consideration in the case of in-order runahead execution.

C. Explaining differences in reuse between in- and out-of-order runahead

Figure 10 provides an example that illustrates the main cause for the relative differences in improvement achieved by the reuse of pre-execution results for in-order and out-of-order runahead pre-execution. This figure shows a stylized representation of the sequence of code from *astar* starting with the load that causes the single-highest number of entries into runahead execution mode for both in- and out-of-order runahead. The load is shown as operation **A** in Figure 10. Load **A** is quickly followed by a consumer **B** which would cause a stall under in-order execution and would begin runahead execution for the in-order runahead-execution model. Though a couple instructions follow on a dependent chain starting with **A**, independent instruction **C** follows these dependent instructions. Instructions starting with **C** (which is almost always a cache hit) would be able to execute independently under pre-execution, including load **D** which like **A** suffers from frequent long-latency cache misses. In the case of such a miss, a long chain of 23 dependent instructions would all be deferred during pre-execution. However, assuming **E** can be reached during runahead, this instruction begins a sequence of independent instructions all of which can execute independently during pre-execution.

In both in- and out-of-order pre-execution models, the relatively short chain from **C**$\rightarrow$**D** and the long sequence starting with **E** will execute and produce correct results. In-order execution with runahead reuse will exploit both of these sequences of precomputation, accelerating execution by skipping the already pre-executed instructions. However, the benefit during out-of-order execution is not nearly as significant. Because normal execution will restart with the load **A**, the serial sequence from **C**$\rightarrow$**D** will be processed sequentially once normal execution has resumed, and reuse in this case will provide a slight benefit. Unfortunately for reuse, the sequence starting with **E** does not always provide similar potential for speedup. Instead, in out-of-order pre-execution models, once normal execution has resumed, program instructions starting with **E** will be loaded into reservation station entries even while the chain of instructions dependent on **D** are being issued and executed. Thus, the sequence starting with **E** can be entirely overlapped with the chain starting with **D** by the normal out-of-order model of execution alone. In this real benchmark example, reuse of the sequence starting with **E** in the out-of-order execution model provides no additional benefit over the typical out-of-order runahead model.

D. Examining the effects of architectural variations

All the previous experiments were performed using the parameters specified in Table I which were selected for comparison with [10]. To further evaluate the potential for result reuse for in-order and out-of-order models, two more sets of experiments were performed. In the first, the access time to main memory was varied by increasing its latency to 1000 cycles and decreasing its latency to 100 cycles with all other parameters from Table I remaining the same. The second set of experiments was performed using the original cache and memory latencies while decreasing the instruction issue width from eight to either four, three, or two instructions. For the smaller instruction issue widths, the number of L1 read/write ports was also decreased from four memory accesses per cycle to two. Both of these experiments focused on the set of medium runahead entry benchmarks.

Figure 11 shows the average execution times for models with main memory latency of 100, 500, or 1000 cycles. As the memory latency increases, the benefit from runahead execution

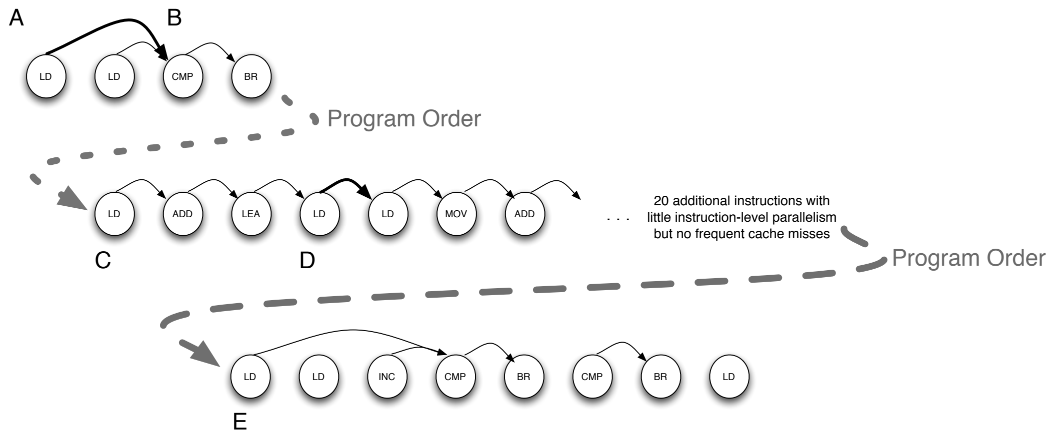

Fig. 10. Example from *astar* of the most frequent runahead-initiating cache-missing load

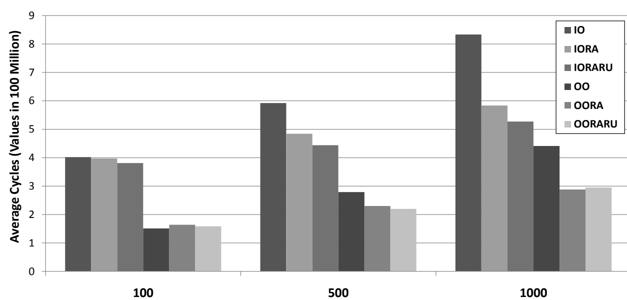

Fig. 11. Average execution cycles for various memory latencies

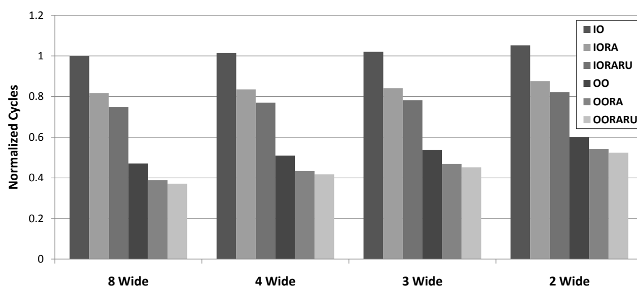

Fig. 12. Normalized execution time for various issue widths

also increases. However, the difference between the average execution times between runahead with and without reuse decrease as the latency goes up for out-of-order models. For the in-order model, the exact opposite behavior is seen with an increase in improvement for runahead with reuse in comparison to traditional runahead. For the 100 latency runs, runahead pre-execution in the out-of-order model actually under performs the baseline out-of-order processor model. This occurs because the out-of-order model is capable of keeping the processor relatively busy with its 128 instruction window. The pipeline flush at the end of runahead mode needlessly hurts the performance of out-of-order runahead model in this case. Another unusual behavior occurs in the 1000 cycle latency simulations, where the runahead with reuse model performs slower than standard out-of-order runahead model. This is the result of the differences in behavior between runahead and runahead with reuse described in Section IV-A. In runahead with reuse, the out-of-order instructions waiting on L2 are not marked as dirty but wait in reservation stations until they are ready. In the basic runahead mode, such instructions would simply be deferred. This difference caused only a minuscule difference in the out-of-order models with a memory latency of 500 cycles. If runahead with reuse had behaved the same as traditional runahead, the slowdown would not have been seen between the two models in the 1000 cycle memory latency case instead they would have appeared identical. As described in Section IV-A, this behavioral difference was in keeping with the setup of in-order runahead and in-order runahead with reuse where significant benefit is obtained by waiting for L2 hits when reusing results.

Experiments were also performed by varying the instruction window sizes from the aggressive out-of-order eight instruction wide pipeline used in [10] all the way down to a modest two instruction wide pipeline. As can be seen in Figure 12, when the instruction width decreases, the number of cycles required to execute the program increases as expected. However, the out-of-order pipeline, with its greater ability to find and exploit instruction-level parallelism, suffers more from the limited issue widths than the in-order pipeline. Additionally, as the pipeline width narrows, the overall difference in performance between the baseline out-of-order and the runahead with reuse models shrink.

In general, for the medium runahead entry set of benchmarks, the benefit from both runahead and runahead with result reuse increases with increasing memory latency for the in-order models. This is to be expected, as the greater the latency to main memory, the greater the benefit from overlapping main memory accesses and the more time available during pre-execution to execute independent instructions. In the out-of-order models, the benefit from runahead alone increases as the memory latency increases, but the benefit from result reuse

actually decreases. This is not too surprising, as the benefit from reuse in the out-of-order case was already minimal. The normal out-of-order execution mechanism alone hides much of the latency of instructions that can be skipped due to pre-execution. In case of the out-of-order models with a longer latency memory, the slight benefit to result reuse is completely outweighed by the benefit from overlapping memory accesses.

For both the in-order and out-of-order models, the benefit from runahead and runahead with result reuse decreases slightly as the issue width shrinks from eight to two instructions per cycle. This is as expected, as in both models, a lesser degree of runahead can be achieved because fewer instructions are issued each cycle. However, the relative decrease in benefit is slightly larger for the out-of-order model.

V. Conclusions

While the benefits of reusing the results of pre-execution provides little speedup for out-of-order runahead systems, similar reuse does provide much greater speedups for in-order systems. Similar to results from previous work [10], result reuse produced during runahead pre-execution mode on an out-of-order processor was shown to provide on average only a 1.03 X speedup over typical runahead alone. However, for an in-order processor model, this result reuse achieves, on average, a 1.09 X speedup over traditional in-order runahead.

While the runahead with reuse provided only a modest overall speedup of 1.09 X, there was a large degree of variation between speedups achieved for the individual benchmarks used in the simulations. These benchmarks were broken down into three distinct categories with the high runahead entry benchmarks experiencing a collective speedup of 1.17 X. Closer examination illustrated that the degree of speedup for reuse for a given benchmark directly correlates to the amount of time spent in runahead mode.

Therefore, even though reuse has little to no impact on out-of-order processors, this result cannot be used to judge the effectiveness of reuse for in-order pipelines. Rather, in-order runahead processors require separate analysis of reuse's effectiveness. Our analysis show that reuse can be beneficial to the in-order runahead systems as a whole, and for certain categories of programs these benefits can provide speedups that allow them to match or outperform a basic out-of-order processors.

References

[1] W. A. Wulf and S. A. McKee, "Hitting the memory wall: Implications of the obvious," *SIGARCH Computer Architecture News*, vol. 23, no. 1, pp. 20–24, March 1995.

[2] S. A. McKee, "Reflections on the memory wall," in *Proceedings of the 1st Conference on Computing Frontiers*, no. 162-167, April 2004.

[3] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, May 1981, pp. 81–88.

[4] 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*, May 1990, pp. 364–373.

[5] P. P. Chang, W. Y. Chen, S. A. Mahlke, and W. W. Hwu, "Comparing static and dynamic code scheduling for multiple-instruction-issue processors," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, November 1191, pp. 25–33.

[6] R. D. Barnes, J. Sias, E. Nystrom, and W. W. Hwu, "EPIC's future: exploring the space between in- and out-of-order," in *Proceedings of the 3rd Workshop on EPIC Architectures and Compiler Technology*, March 2004.

[7] J. W. Sias, S. Ueng, G. A. Kent, I. M. Steiner, E. M. Nystrom, and W. W. Hwu, "Field-testing IMPACT EPIC research results in Itanium 2," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004, pp. 26–37.

[8] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *Proceedings of the 11th International Conference on Supercomputing*, 1997, pp. 68–75.

[9] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead execution: an alternative to very large instruction windows for out-of-order processors," in *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture.*, February 2003, pp. 129–140.

[10] O. Mutlu, H. Kim, J. Stark, and Y. N. Patt, "On reusing the results of pre-executed instructions in a runahead execution processor," *Computer Architecture Letters*, vol. 4, no. 1, pp. 2–5, January 2005.

[11] T. K. Prakash and L. Peng, "Performance characterization of SPEC CPU2006 benchmarks on Intel Core 2 Duo processor," *ISAST Transactions on Computer and Software Engineering*, vol. 2, no. 1, pp. 26–41, 2008.

[12] A. Glew, "MLP yes! ILP no!" in *Proceedings of the ASPLOS Wild and Crazy Idea Session '98*, October 1998.

[13] "2nd generation Intel® Core™ processor family desktop," Intel Corporation, Tech. Rep. 324641-001, January 2011.

[14] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd, "Power7: IBM's next-generation server processor," in *IEEE Micro*, vol. 30, no. 2, March-April 2010, pp. 7–15.

[15] T. Karkhanis and J. Smith, "A day in the life of a cache miss," in *Proceedings of the 2nd Annual Workshop on Memory Performance Issues*, 2002.

[16] O. Mutlu, K. Hyesoon, and Y. N. Patt, "Efficient runahead execution: Power-efficient memory latency tolerance," *IEEE Micro*, vol. 26, no. 1, pp. 10–20, January-February 2006.

[17] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual flow pipelines," *ACM SIGARCH Computer Architecture News*, vol. 32, no. 5, pp. 107–119, December 2004.

[18] R. D. Barnes, J. W. Sias, E. M. Nystrom, S. J. Patel, J. Navarro, and W. W. Hwu, "Beating in-order stalls with 'flea-flicker' two-pass pipelining," *IEEE Transactions on Computers*, vol. 55, no. 1, pp. 18–33, January 2006.

[19] H. Zhou, "Dual-core execution: Building a highly scalable single-thread instruction window," in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, September 2005, pp. 231–242.

[20] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, "Decoupled software pipelining with the synchronization array," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, September 2004, pp. 177–188.

[21] R. D. Barnes, S. Ryoo, and W. W. Hwu, "Tolerating cache-miss latency with multipass pipelines," *IEEE Micro*, vol. 26, no. 1, pp. 40–47, January-February 2006.

[22] A. Hilton, S. Nagarakatte, and A. Roth, "iCFP: Tolerating all-level cache misses in in-order processors," in *Proceedings of IEEE 15th International Symposium on High Performance Computer Architecutre*, February 2009, pp. 431–442.

[23] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Rock: A high-performance SPARC CMT processor," *IEEE Micro*, vol. 29, no. 2, pp. 6–16, March-April 2009.

[24] J. M. Tendler, J. S. Dodson, J. Fields, H. Le, and B. Sinharoy, "POWER4 system microarchitecture," *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5–25, January 2002.

[25] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002, pp. 45–57.