On Reusing the Results of Pre-Executed Instructions in a Runahead Execution Processor

Onur Mutlu*, Hyesoon Kim*, Jared Stark‡, and Yale N. Patt*

*Department of Electrical and Computer Engineering
University of Texas at Austin
{onur,hyesoon,patt}@ece.utexas.edu

‡Microprocessor Research Labs
Intel Corporation
jared.w.stark@intel.com

***Abstract*— Previous research on runahead execution took it for granted as a prefetch-only technique. Even though the results of 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. This paper evaluates the effect of reusing the results of pre-executed instructions on performance. We find that, even with an ideal scheme, it is not worthwhile to reuse the results of pre-executed instructions. Our analysis provides insights into why result reuse does not provide significant performance improvement in runahead processors and concludes that runahead execution should be employed as a prefetching mechanism rather than a full-blown prefetching/result-reuse mechanism.**

I. Introduction

Runahead execution is a technique that improves processor performance by pre-executing the running application instead of stalling the processor when a long-latency cache miss occurs. Previous research has shown that this technique significantly improves the performance of high-performance processors in the face of long main memory latencies. Previous papers on runahead execution took it for granted as a prefetch-only technique [3], [7]. 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. Discarding the instruction results generated during runahead mode, as opposed to buffering them and reusing them in normal mode, is very appealing due to two reasons. First, it simplifies the implementation of a runahead processor. Any attempt at reusing the results generated during runahead mode needs to provide storage for those results and needs to provide a mechanism through which those results are incorporated into the processor state (register file and memory) during normal mode. These requirements increase the area, design complexity, and probably the peak power consumption of the runahead processor.

Second, discarding the instruction results generated during runahead mode decreases the verification complexity of a runahead processor. If results of instructions are not reused, runahead execution does not require any correctness guarantees, since it is purely speculative (i.e., it never updates the architectural state of the processor). Reusing instruction results requires correctness guarantees in the form of mechanisms that make sure that the reused values used to update the architectural state are correct. This requirement is likely to increase the design verification time of a runahead processor.

Manuscript submitted: 17 Nov. 2004. Manuscript accepted: 5 Jan. 2005. Final manuscript received: 14 Jan. 2005.

On the other hand, reuse of instruction results is also appealing in two ways. First, it has the potential to increase performance, since instructions that are valid during runahead mode do not need to be executed during normal mode and therefore they do not need to consume processor resources during normal mode. This frees up space in the processor for other instructions and possibly enables the processor to make faster progress through the instruction stream. Second, the reuse of instruction results increases the efficiency of a runahead processor. An aggressive reuse scheme reduces the total number of instructions executed by a runahead processor, thereby increasing its efficiency.

To quantify the advantages of the reuse of instruction results generated during runahead mode, this paper examines the effects of reuse on the performance of a runahead processor. If the reuse of results could improve the performance of a runahead processor significantly, perhaps it would be worthwhile to research techniques for implementing reuse.

II. Evaluation

We consider the hypothetical *ideal reuse* model to assess the performance potential of runahead result reuse. In this mechanism, if an instruction was valid (correctly-executed) during runahead mode, it consumes no processor resources (including fetch bandwidth) during normal mode and updates the architectural state of the processor with the value it computed during runahead mode. Note that this mechanism is not implementable. We only simulate it to get an upper bound on the performance improvement that can be achieved.

In the *ideal reuse* mechanism, the instructions pseudo-retired during runahead mode write their results and INV status into a FIFO reuse queue, the size of which is sufficiently large (64K entries) in our experiments. At the end of a runahead period, the reuse queue contains a trace of all instructions pseudo-retired in the runahead period along with their program counters, results, and INV status. During normal mode, the simulator reads instructions from this queue, starting with the first instruction pseudo-retired during runahead mode. If the processor encounters an INV instruction in the queue, it is inserted into the pipeline to be executed, since the result for an INV instruction was not generated during runahead mode. If the processor encounters a valid instruction, the instruction is not inserted into the pipeline, but the simulator makes sure that its results are correctly incorporated into the architectural state at the right time, without any latency cost. Hence, the processor is able to fetch a fetch-width worth of INV instructions from the reuse queue each cycle, regardless of the number of intervening valid instructions. If an INV branch that was fetched from the reuse queue is found out

to be mispredicted after it is executed in normal mode, the reuse queue is flushed and the fetch engine is redirected to the correct next instruction after the mispredicted INV branch. An INV branch is mispredicted if the program counter of the next instruction fetched from the reuse queue does not match the address of next instruction as calculated by the execution of the branch. When the reuse queue is empty, the simulator fetches from the I-cache just like in a normal processor.[1] As an optimization, if the processor enters runahead mode again while fetching from the reuse queue, the processor continues fetching from the reuse queue. Therefore, valid instructions that are executed in the previous runahead period do not need to be executed again in the next entry into runahead mode. A runahead period overlapping with a previous runahead period can thus reuse the instruction results generated in the previous period. This allows for further progress in the instruction-stream during runahead mode.

A. Simulation Methodology

We use an execution-driven simulator that models a superscalar, out-of-order processor implementing the Alpha ISA. The machine parameters are listed in Table I. An aggressive state-of-the-art stream-based prefetcher similar to the one described in [11] is also employed in the baseline processor. All experiments were performed using the SPEC 2000 Integer (INT) and Floating Point (FP) benchmarks. INT benchmarks are run to completion with a reduced input set [5].[2] FP benchmarks are simulated using the reference input set. The initialization portion is skipped for each FP benchmark and simulation is performed for the next 250 million instructions.

TABLE I

BASELINE PROCESSOR CONFIGURATION.

Front End	64KB, 4-way I-cache; 8-wide fetch, decode, rename; 64K-entry gshare/PAs hybrid branch pred.; min. 20-cycle mispred. penalty
Execution Core	128-entry reorder buffer; 8 all-purpose functional units, inst. latencies: IMUL(8),FMUL(4),FCVT(4),FCMP(4),FADD(4),FDIV(16),others(1); 64-entry store buffer, stores don't block the reorder buffer
Caches	64KB, 4-way, 2-cycle L1 D-cache, 128 L1 MSHRs, 4 ld/st per cycle; 1MB, 32-way, 10-cycle unified L2, 1 ld/st per cycle; 128 L2 MSHRs all caches have LRU replacement and 64B line size; 1-cycle AGEN
Memory	500-cycle min. latency; 32 banks; 32B-wide, split-trans. core-to-mem. bus at 4:1 freq. ratio; conflicts, bandwidth, and queueing modeled
Prefetcher	Stream-based [11]; 32 stream buffers; can stay 64 cache lines ahead

III. Performance Potential of Result Reuse

Figure 1 shows the IPC of four processors for each benchmark: from left to right, a processor with no prefetcher, the baseline processor, the baseline processor with runahead, and the baseline processor with runahead and ideal reuse. On average, adding runahead execution to the baseline processor increases the baseline IPC by 22.64%. Adding ideal reuse to the runahead processor improves the average IPC by an additional 1.64%. Unfortunately, such a small performance improvement, even with an ideal mechanism, probably does not justify the cost and complexity of adding a reuse mechanism to a runahead execution processor.

[1]Note that this mechanism does not allow the reuse of results generated by instructions executed on the wrong path during runahead mode. Section III-B examines the impact of *ideal reuse* on a processor with perfect branch prediction, which gives the ultimate upper bound on the performance achievable by reusing the results of *all* valid instructions pre-executed in runahead mode.

[2]We also simulated the ideal reuse mechanism with the reference input set for the INT benchmarks and larger L2 caches. These are not presented due to space limitations, but they lead to the same conclusions described in Section III.

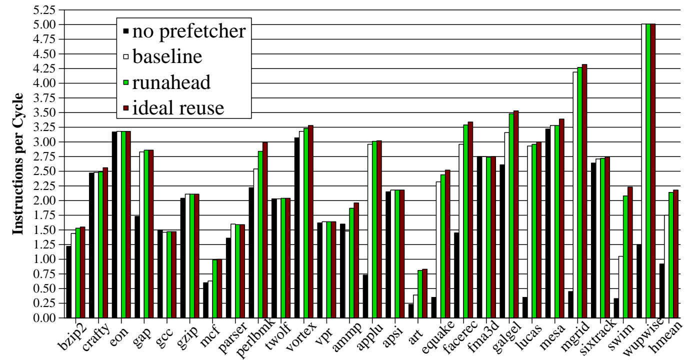

Fig. 1. IPC of the baseline processor with runahead and ideal reuse.

A. Why Does Reuse Not Increase Performance Significantly?

We analyzed why ideal result reuse does not increase the IPC of a runahead execution processor significantly. We identify four reasons as to why this is the case.

First, the percentage of retired instructions that are ideally reused is very low in many benchmarks. Result reuse can only be effective if it significantly reduces the number of retired instructions by eliminating those that are executed in runahead mode. Figure 2 shows the breakdown of retired instructions based on where they were fetched from. An instruction that is "Reuse queue fetch - ideally reused" does not consume processor resources in normal mode. All other instructions need to be executed as described in Section II. Figure 2 shows that only 8.5% of the instructions are eliminated (ideally reused) during normal mode.[3] The percentage of ideally reused instructions is low because: (1) For many benchmarks, the processor spends a small amount of time during runahead mode, which means only a small number of instructions are pre-executed in runahead mode. Hence, there are not a lot of opportunities for reuse. (2) The results of instructions after a mispredicted INV branch cannot be reused. This limits the number of reused results especially in integer benchmarks, where 43% of all instructions pseudo-retired during runahead mode are after a mispredicted INV branch. Unfortunately, even in FP benchmarks, where only 5% of instructions pseudo-retired during runahead mode are after a mispredicted INV branch, ideal reuse does not result in a large IPC increase due to the reasons explained in the following paragraphs.

Second, and more importantly, eliminating an instruction does not guarantee increased performance because eliminating the instruction may not reduce the critical path of program execution. The processor still needs to execute the instructions that cannot be eliminated because they were INV in runahead execution. After eliminating the valid instructions, instructions that are INV become the critical path during normal mode execution.[4] If this INV critical path is longer than or as long as the critical path of the eliminated valid instructions, we should not expect much performance gain unless the processor is limited by execution bandwidth. Our analysis of the code shows the execution of valid instructions after runahead mode

[3]Only in perlbmk, ammp, art, equake, lucas, and swim does the percentage of eliminated instructions exceed 10%. With the exception of lucas, these are the benchmarks that see the largest IPC improvements over the runahead processor with the *ideal reuse* mechanism.

[4]In some cases they were already on the critical path, regardless of the elimination of valid instructions.

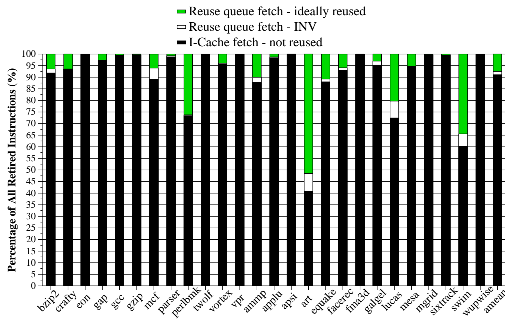

Fig. 2. Classification of retired instructions in the ideal reuse model.

is actually much faster than the execution of INV instructions, in general. Instructions that were valid in runahead mode can be executed much faster in normal mode because they do not incur any L1 or L2 cache misses in normal mode.[5] In contrast, instructions that were INV can and do incur L1 and L2 misses during normal mode, making them more expensive to execute. We find that the average execution time of a valid instruction is only 2.48 cycles after runahead mode, whereas the average execution time of an INV instruction is 14.58 cycles. Thus, ideal reuse eliminates valid instructions that do not take long to execute. Therefore, it cannot reduce the critical path of the program significantly and results in small performance improvements.[6] Table II provides supporting data showing that although most of the instructions pre-executed in runahead mode are valid (row 6), an important percentage of INV instructions incur very long latency L2 misses and relatively long latency L1 misses in normal mode (rows 7-8) and therefore take much longer to execute than valid instructions (rows 12-13).

TABLE II

RUNAHEAD EXECUTION STATISTICS RELATED TO IDEAL REUSE.

Row	Statistic	INT	FP	Average
1	Percent cycles in runahead mode	13.7%	27.1%	21.2%
2	Cycles per runahead (RA) period	474	436	454
3	L2 misses captured per RA period	1.45	3.4	2.5
4	Pseudo-retired inst. per RA period	1240	1169	1201
5	Correct-path inst. per RA period	712 (57%)	1105 (95%)	923 (77%)
6	INV inst. per RA period	544 (44%)	264 (23%)	393 (33%)
7	% of INV/valid that are L2 miss	1.4%/0%	2.1%/0%	1.9%/0%
8	% of INV/valid that are D-cache miss	3.0%/0%	2.8%/0%	2.9%/0%
9	% of INV/valid that are D-cache hit	38%/38%	36%/44%	36%/43%
10	% of INV/valid inst. that are FP	0.8%/0.4%	60%/25%	43%/20%
11	% of INV/valid that are 1-cycle	59%/61%	2%/31%	18%/37%
12	Avg. latency of valid inst. (after RA)	1.78 cyc	2.66 cyc	2.48 cyc
13	Avg. latency of INV inst. (after RA)	10.16 cyc	16.31 cyc	14.58 cyc
14	IPC improvement of ideal reuse	0.90%	2.45%	1.64%

Figure 3 shows an example data-flow graph from mcf demonstrating why reuse may not increase performance. In this example, shaded instructions were INV during runahead mode. Numbers by the data-flow arcs show the latency of the instructions. After exit from runahead mode, the second INV load takes 13 cycles because it incurs an L1 data cache miss. Therefore, the INV dependence chain takes 17 cycles to execute. If there is no result reuse, valid instructions take only 6 cycles to execute because there are enough execution resources in the processor core. Hence, eliminating these valid instructions via ideal reuse does not affect the critical path and does not increase performance, even though the number of valid instructions is larger than the number of INV ones.

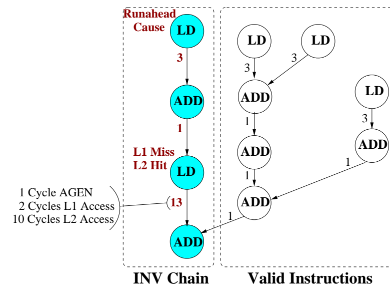

Fig. 3. Example showing why reuse does not increase performance.

Third, if the execution time of a benchmark is dominated by long-latency memory accesses (memory-bound), the performance potential of result reuse is not very high since reuse can only speed up the smaller execution-bound portion. That is, the ideal reuse of valid instructions cannot reduce the number of cycles spent on long-latency memory accesses. Note that this is a special case of Amdahl's Law [2]. If the optimization we apply (reuse) speeds up only a small fraction of the program (execution-bound portion of the program), the performance gain due to that optimization will be small. Unfortunately, memory-bound benchmarks are also the benchmarks that execute many instructions in runahead mode. Art is a very good example of a memory-bound benchmark. We find that, in art, 86% of all cycles are spent waiting for long-latency L2 load misses to be serviced. Hence, even if all other execution cycles are magically eliminated, only a 16% IPC improvement can be achieved. This constitutes a loose upper bound on the IPC improvement that can be attained by the *ideal reuse* scheme. In our simulations, the *ideal reuse* scheme only increases the IPC of art on the runahead processor by 3.3%, even though it eliminates 51% of all retired instructions.

Fourth, a processor that implements ideal reuse enters some performance-degrading short runahead periods that do not exist in the baseline runahead processor. The *ideal reuse* processor reaches an L2-miss load faster than the baseline runahead processor, due to the elimination of valid instructions in normal mode. If there is a prefetch pending for this load, the ideal reuse processor provides less time than the baseline runahead processor for the prefetch to complete before the load is executed (since it reaches the load quicker), which makes the ideal reuse processor more likely to enter runahead mode for the load than the baseline runahead processor. This runahead period is usually very short and results in a pipeline flush without any performance benefit. Such periods, which do not exist in the baseline runahead processor, reduce the performance benefit due to ideal reuse.

B. Effect of Main Memory Latency and Branch Prediction

Memory latency is an important factor on the performance impact of result reuse, since it affects the number of instructions executed during runahead (hence, the number of instructions that can possibly be reused) and it affects how much of a program's total execution time is spent on servicing L2 cache

[5]Except in the rare case when the runahead period was so long that it caused thrashing in the L1 cache.

[6]Note that ideal reuse is more effective in FP benchmarks than in INT benchmarks, because it eliminates some FP dependence chains that take a long time to execute due to the relatively long-latency FP instructions.

misses. Branch prediction accuracy is also an important factor, because it affects the number of *useful* instructions that can be reused (by affecting the number of mispredicted INV branches during runahead mode or by affecting when they occur during runahead mode [7]).

The left graph in Figure 4 shows the average IPC improvement of runahead execution and runahead with ideal reuse over the respective baseline processor for four different memory latencies. As the memory latency becomes shorter, the processor spends less time and executes fewer instructions during runahead mode. This reduces the percentage of retired instructions that are ideally reused in a program. For all four memory latencies, the IPC improvement of ideal reuse is not very promising.

The right graph in Figure 4 shows the average IPC improvement of runahead execution and runahead with ideal reuse over the respective baseline processor with perfect branch prediction for the same four memory latencies. Ideal reuse on a processor with perfect branch prediction shows more performance potential for INT benchmarks than ideal reuse on a processor with imperfect branch prediction. On INT benchmarks, with perfect branch prediction, runahead execution by itself improves the IPC of the baseline processor with 500-cycle main memory latency by 18%, whereas ideal reuse improves the IPC of the same processor by 23%. Therefore, even if an ideal reuse mechanism is implemented in a runahead processor, the prefetching benefit of runahead execution would provide most of the performance improvement obtained over the baseline processor. The additional performance improvement due to ideal reuse is still perhaps not large enough to warrant the implementation of a realistic reuse mechanism, even with perfect branch prediction.

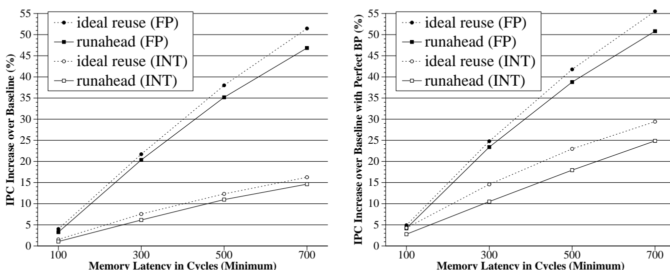

Fig. 4. Effect of memory latency and branch prediction on reuse performance.

IV. Related Work

Several previous papers proposed the use of runahead execution for improving the latency tolerance of microprocessors [3], [7]. The possibility of the reuse of results generated during runahead execution was mentioned by Mutlu et al. [7], but design trade-offs related to it were not discussed or evaluated. Dundas evaluated reusing the branch outcomes generated during runahead mode and found that such reuse did not result in significant performance benefits [4].

Sodani and Sohi proposed a general-purpose mechanism for reusing the results generated by instructions in order to eliminate redundant computations [8]. We pursue the same goal of eliminating redundant computations and possibly instructions by evaluating the reuse of runahead results. Our results are not as promising as what they reported because: (1) memory latencies are much longer in today's processors, which makes the execution of instructions cheap in terms of performance; (2) we evaluated the effect of result reuse in a more special-purpose context, i.e., reusing runahead results.

Srinivasan et al. proposed Continual Flow Pipelines [10] as a mechanism that stores instructions dependent on L2 misses in a large buffer so that they do not occupy scheduler and register file entries for a long time. This mechanism is not a *reuse* mechanism, but it frees up resources so that L2-miss independent instructions can execute early in a CPR processor [1]. Srinivasan et al. found that allowing the execution of L2-miss independent instructions significantly improves the performance of a CPR processor, a result that is seemingly at odds with the results presented in this paper. The differences in results can be due to: (1) the fundamental differences between the CPR and runahead processing paradigms; e.g., a runahead processor *pre-processes* the L2-miss independent instructions whereas a CPR processor does not, which may make the processing of these instructions more costly for the CPR processor; (2) the differences in the evaluated benchmarks and the ISAs; (3) the differences in the modeled instruction latencies (Srinivasan et al. [10] did not specify the modeled instruction latencies). Further research is needed to understand these differences.

Value prediction [6] is a technique that may overcome some of the limitations of reuse [9]. Therefore, combining runahead execution with value prediction may have more potential and we are examining this in our current work.

V. Conclusion

We examined the performance impact of reusing the results of instructions pre-executed in runahead mode. Even with an ideal reuse mechanism, along with perfect branch prediction, reuse of results does not show significant performance benefits. We provided insights into why this is the case. We conclude that runahead execution should be employed as a prefetching mechanism without reuse, since reuse does not provide significant performance benefits while it very likely significantly increases the implementation cost and complexity.

References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *MICRO-36*, pages 423–434, 2003.

[2] G. M. Amdahl. Validity of the single-processor approach to achieving large scale computing capabilities. In *AFIPS Conference Proceedings*, pages 483–485, 1967.

[3] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *ICS-11*, 1997.

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

[5] A. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1, June 2002.

[6] M. H. Lipasti, C. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *ASPLOS-7*, pages 226–237, 1996.

[7] 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 *HPCA-9*, pages 129–140, 2003.

[8] A. Sodani and G. S. Sohi. Dynamic instruction reuse. In *ISCA-24*, pages 194–205, 1997.

[9] A. Sodani and G. S. Sohi. Understanding the differences between value prediction and instruction reuse. In *MICRO-31*, pages 205–215, 1998.

[10] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual fbw pipelines. In *ASPLOS-11*, pages 107–119, 2004.

[11] J. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Technical White Paper*, Oct. 2001.