Enhancing MLP: Runahead Execution and Related Techniques

Brian Greskamp Pablo Montesinos Abhishek Tiwari
i-acoma group
Department of Computer Science
University of Illinois Urbana–Champaign
201 N. Goodwin Ave.
Urbana, IL 61801
E-mail: {greskamp,pmontesi,atiwari}@cs.uiuc.edu

1. Introduction

The growing memory wall[1] makes speedups increasingly difficult to achieve on applications that exhibit difficult-to-predict memory access patterns. The problem is that although modern processors provide multiple high-bandwidth execution units, applications that experience frequent cache misses are only executed with high IPC in the periods between misses. As main memory latencies increase from 200 cycles to 1000 cycles and beyond, microarchitects must scramble for new ways to hide as much of this latency as possible.

It is now widely understood that one solution lies in maximizing the number of simultaneous outstanding cache misses, known as Memory-Level Parallelism (MLP). Prefetching, long instruction windows, and runahead execution have all been proposed as a means of enhanching MLP. In this paper, we will discuss examples of these three techniques, noting the performance and design complexity of each. We focus in particular on a prefetching/pre-execution scheme known as *runahead execution*, which pre-executes instructions following a cache miss—when the processor would otherwise stall—in hopes of generating useful prefetches. We compare performance and design complexity of several runahead designs with those of alternative solutions. We argue that runahead is a simple, elegant, and effective technique for increasing MLP.

2. Background

Memory Level Parallelism, or MLP, is a metric that simplifies understanding of memory's limiting effect on performance. We use the definiton from Chou *et al.* [21]: *The average MLP is equal to the average number of outstanding useful long-latency misses when there is at least one long-latency miss pending*. By 'useful misses', we mean only those which bring data that is read by the procesor before being evicted from the cache.

In current architectures, once a single main memory miss is outstanding, structural stalls are usually imminent. Delaying or eliminating those stalls allows more misses to be initiated, increasing MLP if those misses bring useful data into the cache. Clearly, maximizing the number of long latency misses outstanding during each stall (the MLP) is beneficial because it minimizes the total number of stalls. In fact, in the limit of increasing memory latency, MLP comes to dominate performance [8]. In such cases, performance can be approximated with an *epoch model* [21] as shown in Figure 1.

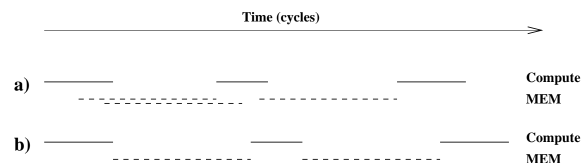

Figure 1. Overlap of computation and main memory accesses (a) and the epoch model of the same execution (b)

The epoch model describes the performance of small-window superscalars; intervals of computation alternate with intervals of memory access, and the overlap between computation and memory access is vanishingly small. Under the epoch model, the key to increasing performance is to perform as many memory operations as possible in each *memory* interval. According to [4], a superscalar with window size 64 and 200 cycle memory latency realizes an MLP of only 1.13 to 1.38 on database applications.

Before discussing runahead execution, we survey other techniques for enhancing MLP. First, we discuss large in-

[1] One of only three man-made structures visible from space

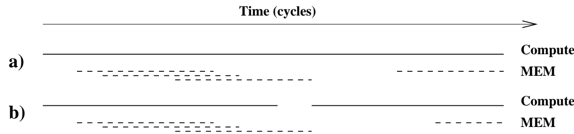

Figure 2. If the window size exceeds the memory latency, L2 misses do not stall the processor

struction windows, a natural enhancement to out-of-order processors. Next, we survey work that uses prefetching threads to reduce L2 miss frequency. We show how each method overlaps more main memory misses and cite example implementations and performance numbers for each.

2.1. Large Instruction Windows

In the context of superscalar processors, perhaps the most obvious solution to the MLP problem is to simply enlarge the instruction window[2]. Window stalls due to L2 misses can be eliminated entirely with a large enough window. Even if not large enough to completely hide the stalls, an enlarged window can still provide increased MLP and greatly boost overall performance. Figure 2 illustrates (a) a window that completely hides misses and (b) a window that allows overlapping more misses despite being too small to hide the full main memory latency.

Unfortunately, the microarchitectural structures that support aggressive out-of-order execution (*eg.* large register file, reorder buffer, and scheduling logic), have all proven difficult to scale [17]. Here we survey methods of expanding both the ROB and the scheduling window, along with the performance impact of each. It should be noted that other structures such as MSHRs and load/store queues must also scale to support large instruction windows, but those details are not discussed here.

2.1.1. Expanding the ROB

Karkhanis and Smith [18] showed that decoupling the ROB and scheduler can reduce the frequency of structural window stalls. A "decoupled" ROB is one that has more entries than the scheduler (*ie.* the ROB and scheduler are separate structures). Karkhanis and Smith performed trace-based simulation, assuming a memory latency of 1000 cycles. They modeled a perfect L1 cache but inserted misses at random (non-overlapping) locations in the dynamic instruction stream. They reported that for an unlimited ROB and 1000 cycle memory latency, each miss in SPECint 2000 accumulates about 30 dependent instructions in the scheduler before the missing load returns. In other words, a missing load causes a relatively small number of dependent instructions to pile up in the scheduler compared with the number of ROB entries consumed by subsequent instructions. Karkhanis and Smith concluded that increasing the size of the ROB and register file alone could yield large performance improvements without resorting to exotic, large schedulers.

Checkpoint Processing and Recovery (CPR) [1] by Akkary *et. al* is one example of a large-ROB scheme. Actually, the CPR scheme eliminates the ROB entirely, replacing it with a set of microarchitectural checkpoints taken at low confidence branches. The checkpoint-based microarchitecture eliminates the need for large physical register files. In an ROB-based scheme, registers are reclaimed only when their writing instruction reaches the head of the ROB. On the other hand, CPR allows registers to be reclaimed as soon as there are no more readers.

To justify CPR, the authors performed simulations of an idealized processor with a 2048 entry ROB. On that processor, they found that a 256 entry scheduler performed very nearly as well as a full 2048 entry scheduler. They concluded that a scalable alternative to the ROB was required, but that a scheduler as small as 128 entries would be acceptable.

2.1.2. Large Schedulers

While [1] and [18] suggest that the size of the scheduling window is not so critical, that conclusion appears to be highly benchmark-dependent. In [4], enlarging the scheduler from 64 to 128 entries on an aggressive processor with a $2K$ entry ROB increased MLP by roughly 10% on a collection of enterprise benchmarks. In [1], the performance difference between a 256 and 2048 entry scheduler on an idealised superscalar is indistinguishable for SPECint and SPECweb. The same scheduler configurations in [19] differ in performance by only 4% for SPECint and 7% for SPECweb. However, in the latter paper, the larger scheduler delivers a 22% speedup on SPECfp.

Since larger schedulers can indeed speedup some applications, the CPR authors have recently proposed Continual Flow Pipelines (CFP) [19], an extension to CPR that provides a larger scheduler and generates speedups over the base CPR system for some benchmarks. CFP operates by moving load-miss dependent instructions out of the scheduler and into a *slice buffer*. As instructions are moved into the slice buffer, ready register values are read and carried to the buffer along with the instruction. This allows the processor to free the instruction's physical registers. When deferred instructions are reintroduced into the pipeline, a *back-end renamer* reallocates new physical registers for them. Since instructions in the slice buffer do not consume registers, only a modest-sized physical register file is required.

Similar in spirit to CFP, Lebeck *et al.* proposed the Wait-

[2]The set of instructions fetched but not yet completed

ing Instruction Buffer (WIB) [11], a queue in which miss-dependent instructions are temporarily stored until they become ready. Like CFP, the WIB allows the scheduler to remain small (eg. 32 entries) by preventing it from filling with miss dependent instructions. In the WIB design, load misses distribute a 'pretend ready' tag to their dependent instructions in the scheduler. Those instructions that have all of their operands either ready or 'pretend ready' are woken up. When an instruction with at least one 'pretend ready' argument is selected, it is issued to the WIB instead of a functional unit. Each instruction in the WIB is marked with the earliest dynamic load on which it depends so that it can be re-dispatched into the scheduler when the load returns.

With a WIB, deferred instructions still consume physical registers. Therefore, a very large physical register file is required. Fortunately, hierarchical register files can satisfy that requirement, albeit with some additional complexity.

2.2. Prefetch Threads

Instead of proposing microarchitectural enhancements, some researchers have exploited existing SMT support to address the MLP problem. The basic idea is to use otherwise idle hardware threads to execute prefetches for the main thread. Here we discuss two recent proposals.

Zilles and Sohi [22] observe that the majority of misses are caused by a small number of static instructions. They begin by building backward slices to compute the addresses of these 'delinquent' loads. The slices are then spawned as separate hardware threads at appropriate points in the main program's execution. Generally, a spawn point is hoisted as high as possible. If the spawn point is early enough then the slice will complete execution and initiate a miss before the main program actually requires the data. Unfortunately, the slices tend to be so large that they do not execute any faster than the main program.

To make the slices execute faster, Zilles and Sohi optimize them in ways that might not always be correct. By applying these aggressive optimizations, they are able to heighth-reduce the slices. Since they never allow these *speculative* slices to affect architectural state, the main thread remains correct. They report that speculative prefetch slices are able to eliminate 33% of cache misses from SPECint 2000.

Collins *et al.* [5] simulate prefetch threads on an SMT processor using the Itanium ISA. They confirm the observation of Zilles and Sohi [22] that a few delinquent static loads cause a majority of cache misses. Like Zilles, Collins manually identifies delinquent loads using memory access profiling and constructs slices to compute those load addresses. Collins' contribution is the *chaining trigger*, through which one speculative thread spawns another. For example, a thread that has just finished prefetching for one iteration of

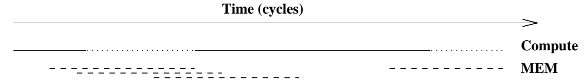

Figure 3. Runahead enhances MLP by pre-executing future misses when the processor would otherwise stall

a loop may spawn another thread to fetch for the next iteration. Unfortunately, Collins' evaluation is incomplete, using only a subset of the SPECint and SPECfp benchmarks, so a direct performance comparison with [22] is not possible.

It is important to note that both Collins and Zilles generated their prefetch threads manually, without formalizing their methods or building an actual compiler. Kim and Yeung [9] have recently proposed a source-to-source prefetch thread extractor and compiler that performs profiling to identify delinquent loads as well as slicing and optimization to generate the actual prefetch code.

Prefetch threads on SMT are attractive because they make use of existing hardware, effectively providing a software-only solution that works quite well. All of the other proposals we discuss require new microarchitectural features. Consequently, prefetch threads are in a sense less complex than the other schemes.

3. Runahead Execution

Runahead execution [2, 7, 20] is an elegant scheme that generates accurate data prefetches by speculatively executing past a long latency cache miss, when the processor would otherwise be stalled. This section discusses several implementations of runahead in detail, but the basic idea for each is the same: Pre-execute miss-independent instructions to generate highly accurate prefetches. Figure 3 shows how runahead increases MLP. The processor would normally experience a window stall where the solid 'compute' line ends. Runahead allows the processor to continue speculatively (dotted line), generating an additional useful prefetch before the window-blocking load returns.

Here we discuss runahead implementations for different microarchitectures: in-order scalar processors, VLIW/EPIC processors, and out-or-order superscalars.

3.1. In-Order Scalar

Dundas *et al.* [7] proposed using the idle cycles caused by a cache miss in an in-order scalar processor to pre-execute future instructions. Their scheme is simple, yet effective: When the processor detects an L1 data cache miss, it records the missing address and checkpoints the register file into a backup register file. It then executes the instructions that follow the miss while waiting for memory.

Clearly, these pre-executed instructions can be dependent upon invalid data, so care must be taken. An `INV` bit is associated with each register and L1 cache line to denote whether the data stored there is invalid. Each type of instruction has rules for propagating the `INV` value:

- An instruction marks its destination register `INV` of any of its source operands are `INV`.
- Load instructions set their target register's `INV` bit when their address register or the target word in the cache is marked `INV`, or when they cause a cache miss.
- Stores do not write any data to cache or memory, but they set the `INV` bit of the target line even if the address register is not `INV` and a cache miss does not occur.
- Conditional branches are executed normally if the condition is not `INV`. If the condition register is `INV`, the processor must trust the branch predictor.

Once the original missing load receives its data, runahead mode is switched off, the register file is restored from its backup and the `INV` bits in the cache and registers are reset.

3.2. EPIC/VLIW

EPIC architectures are also very sensitive to memory stalls. Barnes [2] showed that 38% of Itanium execution cycles are spent in L1 D-cache misses for SPECint, with many of them being due to accesses satisfied in the L2 cache. In an EPIC machine, a cache miss stall will prevent all the following instructions (not only the dependent ones) from issuing until the load is resolved. Compare this situation with an out-of-order processor, where a 5-cycle L2 cache miss is easily hidden by other instructions in the instruction window.

To combat L2 D-cache miss stalls in Itanium, Barnes [2] proposed the *flea-flicker* microarchitecture, a design that composes two in-order sub-pipelines. The microarchitecture consists of an 'advance pipeline' and a 'backup pipeline'. The advance pipeline executes all instructions speculatively. If the dependence check finds one instruction not ready, the advance pipeline does not stall but marks the not-ready instruction and all its dependents as deferred instructions (using **INV** bits), scheduling them to be executed in the backup pipeline. Instructions executing in the backup pipeline stall if any of their operands is not ready, just as in a normal EPIC processor. Only the backup pipeline can change the architectural state.

Two key structures in the flea-flicker microarchitecture are the Coupling Queue (CQ) and the Coupling Result Store (CRS). The CQ is used to send in-order, predecoded, deferred instructions from the advance pipeline to the backup pipeline. The CRS is used to send the results of the pre-executed instructions to the backup pipeline. The backup pipeline is augmented with a merge stage where results arriving from the CRS are bypassed to instructions and written to the architectural register file as appropriate. Other structures required to maintain the correctness and efficiency of the design are the Update Queue (UQ) and a modified ALAT. The UQ sends register writes from the backup pipeline back to the speculative register file in the advance pipeline. The modified ALAT allows the backup pipeline to detect when a load is executed in the advance pipeline without observing a previous (deferred) conflicting store.

The biggest drawback of the flea-flicker microarchitecture is the complexity of the approach. Whereas straightforward runahead implementations for in-order scalar and out-of-order superscalar processors need modest hardware, flea-flicker requires several new hardware structures as well as duplication of the processor pipeline (compare Figures 4 and 5).

3.3. Out of Order

Although out-of-order processors tolerate cache misses better than in-order processors, Mutlu [20] showed that an idealized version of an out-or-order processor still spends most of its time waiting for data from main memory. Mutlu proposed using runahead execution in out-of-order processors, effectively transforming a small, blocking instruction window into a non-blocking one.

An out-of-order runahead processor enters runahead mode when a memory operation misses the L2 cache and becomes the head of the ROB. The processor switches back to normal mode when the data returns from memory. As in the in-order scalar case, the address of the instruction that caused the miss is stored and the register file is checkpointed. Any in-flight or newly fetched instructions are treated as runahead instructions, propagating `INV` bits.

In [20], the propagation of register `INV` values works identically to the original scalar in-order proposal [7]. However, memory operations are handled more carefully than in [7]. Stores set the `INV` bit only when their data or address operands are `INV`, and the results of runahead stores are forwarded to runahead loads. If the store is still within the instruction window when a dependent load issues, the store buffer is used to forward the value and the `INV` bit. Otherwise, the data and `INV` flag are read from a special *runahead cache* [16]. The runahead cache is only accessed by runahead loads and stores and never spills to memory. Any load that misses in both the runahead and normal caches will have its destination marked as `INV`. Figure 5 shows the microarchitectural additions to an out-of-order processor to

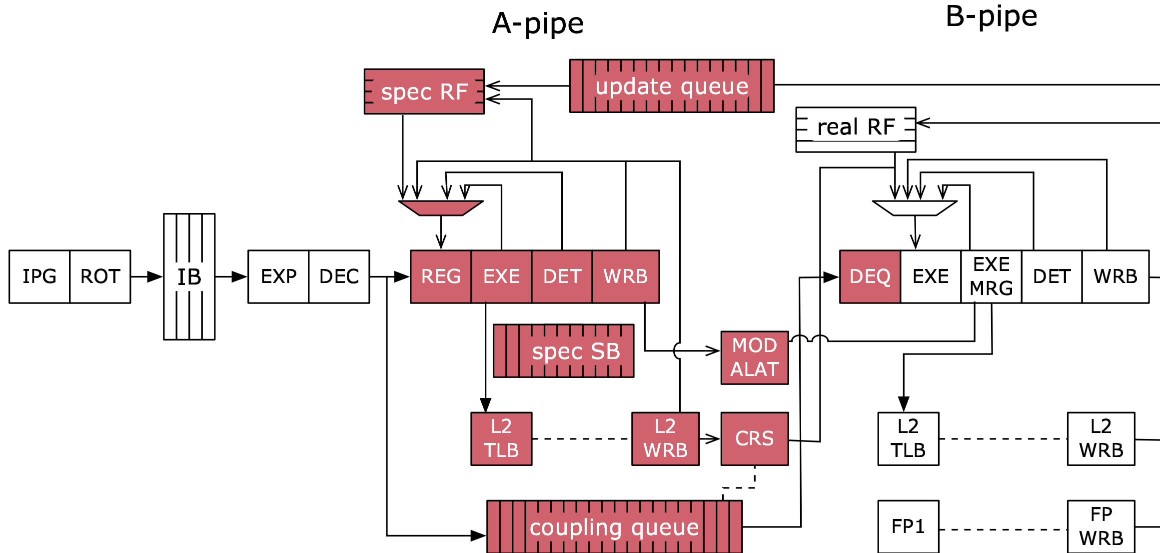

Figure 4. The flea-flicker microarchitecture. Additions to the Itanium pipeline are in red.

support runahead execution. Components with a red stripe have been made aware of `INV` bits, and red components are not present in a normal superscalar. Subjectively, the additional hardware complexity appears reasonable.

Two more recent proposals, Checkpointed Early Load Retirement [10] and CAVA [3], build on [20] by adding value prediction. Correctly predicting the value of a missing load eliminates the need to rollback to a checkpoint when the load returns. The mechanism in [10] is similar to the basic out-of-order scheme described above. Once a long latency load arrives at the head of the ROB, the processor enters *clear mode*. The architectural registers are checkpointed, the load is *early-retired*, and its value is predicted so execution of dependent instructions can proceed. If another load miss occurs while the processor is in clear mode, the hardware decides whether to take another checkpoint depending on the value predictor confidence. When the data from an early-retired load returns, the processor compares it with the prediction for that load. If the prediction was incorrect, the processor rolls back to that load's checkpoint (or the latest preceeding checkpoint if the load was not checkpointed) and continues normal execution. Otherwise, there is no need to roll back.

4. Analysis

In this section, we address some current issues in the design of runahead processors. Firstly, there is a middle ground between a true large-window processor (*ie.* large scheduler and large ROB) and runahead processors. It is possible to build runahead processors that do not throw away results that are correctly computed during runahead mode, and we would like to know whether this is worthwhile for in-order and out-of-order processors. Secondly, runahead proposals have recently begun incorporating value prediction [3, 10]. We would like to know how value prediction improves performance (*ie.* by generating more accurate prefetches or by facilitating reuse) and whether it is worth incorporating. Finally, runahead processors execute more instructions than a traditional processor, so we describe techniques to improve the efficiency of runahead execution. This section considers these issues in turn.

4.1. Runahead vs. Large Windows

Conceptually, runahead execution has much in common with large instruction windows. Both schemes allow the processor to continue executing miss independent instructions after a long-latency miss. The key difference is that runahead execution discards these results rather than incorporating them into the main execution. The advantage of runahead is that it is comparatively simple to implement and verify. A potential disadvantage is that results correctly computed during runahead mode are wasted, with energy and performance consequences. Therefore, the important questions are: (1) What is the performance difference between a runahead processor and a 'true' large-window processor? and (2) Is the additional complexity of large window schemes acceptable in this context?

Firstly, there is some disagreement about the magnitude of the performance difference between large-window and

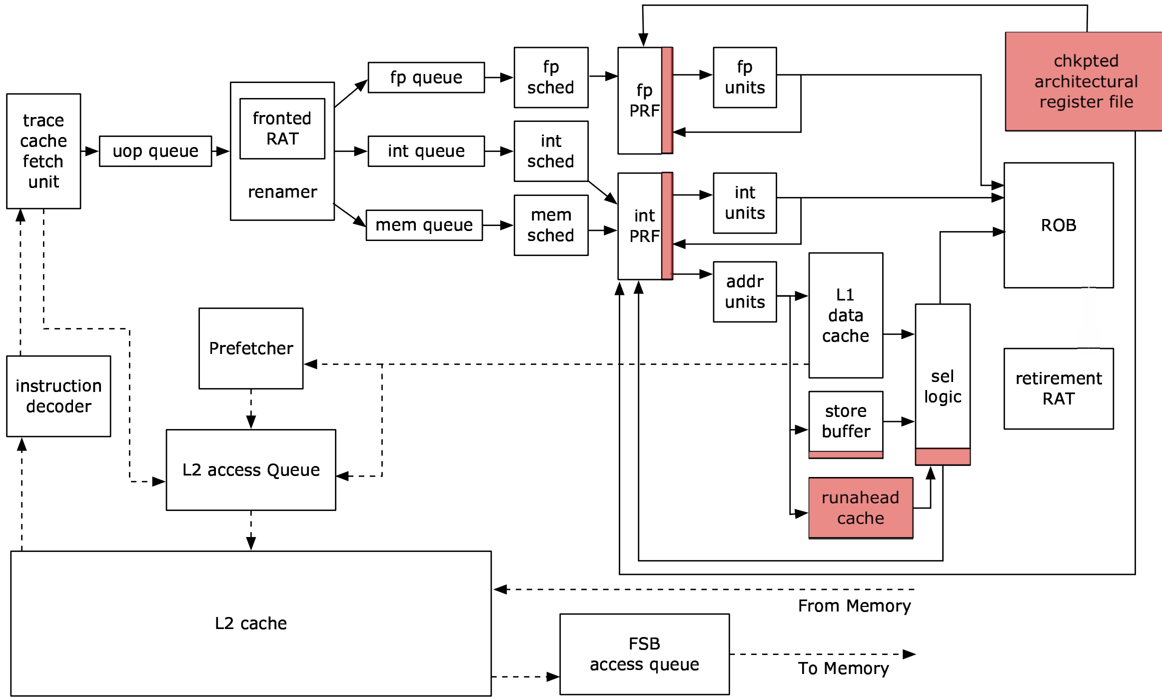

Figure 5. The out-of-order runahead microarchitecture. Additions to a P4-like processor are in solid red. Red stripes mark components that have been made aware of `INV` bits.

runahead processors. The authors of CFP [19] compared their processor against a runahead processor based on CPR and found significant speedups on a range of benchmarks: 18% for SPECfp, 5% for SPECint, and 22% for a CAD workload. Interestingly, they also found that runahead execution measurably *decreased* performance on SPECint and server workloads when compared to the base CPR (effectively a large-ROB) system.

On the other hand, [15] performed a limit study on the impact of 'ideal reuse' in runahead systems and came to a very different conclusion. In that study, the authors added a *reuse buffer* to their runahead simulator (the same simulator as used in [20]). The buffer holds instructions executed during runahead mode along with their results. Upon return to normal execution, results of instuctions that executed correctly in runahead mode are incorporated free-of-cost into the machine state as they are encountered. The ideal reuse model thus provides an upper bound on the performance achievable by a runahead processor with reuse.

Compared to the runahead proposal in [20], ideal reuse gave speedups of only 2% for both SPECfp and SPECint. Although 8.5% of dynamic instructions (averaged over SPECint and SPECfp) were ideally reused, the performance impact was minimal because the reused instructions were not likely to be on the critical path. After exiting runahead mode, instructions that were not `INV` usually have their data available in the cache, so although their results can be reused, the savings is only 2.58 cycles per instruction on average. Instead, it is the instructions that were `INV` during runahead mode that are now on the critical path, taking an average of 14.58 cycles each.

The results in [19] and [20] are apparently in contradiction. A potential explanation hinges on precisely when reused results are available in the ideal reuse model. In the case that a long latency load miss is followed by a chain of independent, short latency instructions as shown in Figure 6, CFP and ideal reuse may behave differently. Figure 7 (b) shows how execution proceeds on CFP. The missing load is moved into the slice buffer and the chain of instructions $2..n$ executes. The load data eventually returns. At that point, instructions $2..n$ have updated the microarchitectural state. Figure 7 (a) shows execution on a runahead architecture with reuse. Although the results for instructions $2..n$ are correctly generated in runahead mode, they are not merged with the execution state until runahead mode terminates. Since each instruction $2..n$ is dependent on its predecessor, it takes $n-1$ cycles to merge the results for those instructions into the execution.

Another interesting case for reuse is made in the flea-flicker [2] microarchitecture. Here the important factor is

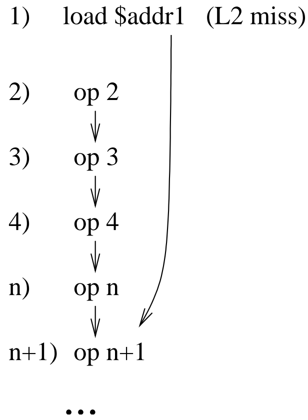

Figure 6. A program to demonstrate the difference between reuse and large windows. Arrows represent data dependences

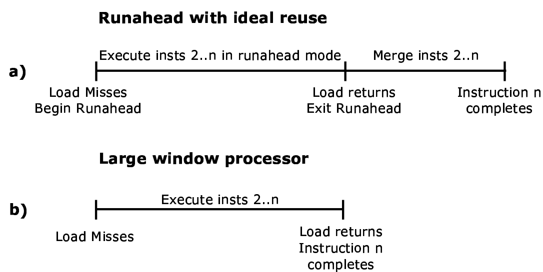

Figure 7. Execution of code segment in Figure 6 on a runahead processor with ideal reuse (a) and a large window processor (b)

that the processor pipeline is in-order. If an instruction uses a load that has not yet returned, the entire processor is stalled. In other words, even L1 misses can hurt badly—this is the motivation for two-pass pipelining. Although many of the loads initiated in the advance pipeline will be in the L1 cache by the time they are processed in the backup pipeline, some may have been evicted back to the L2. In that case, reusing the result from the advance pipeline avoids an L1 miss and potential stall.

Returning to out-of-order processors, the most compelling argument against large instruction windows or reuse is complexity. The allure of runahead is that it is so simple because *all* of the data generated during runahead execution is speculative and guaranteed to *never* enter architectural state.

Additionally, a runahead processor is able to exploit MLP in cases where a large window processor can not unless it speculates very aggressively. An example given in [4] is the handling of synchronization instructions (*i.e.* MEM barrier or read-modify-write). A straightforward superscalar implementation will require the processor pipeline to drain before the serializing instruction can issue and the frequency of these instructions is quite high for some benchmarks (*eg.* 0.6% of dynamic instructions for SPECjbb). A large-window processor must therefore implement aggressive memory speculation features to avoid pipeline flushes that effectively truncate the window. On the other hand, a processor in runahead mode does not have to respect ordering of synchronization operations since all runahead computation is speculative. As a result, runahead processors can extract more MLP from lock-intensive code without requiring complicated microarchitecture support for memory speculation.

4.2. Value Prediction

CAVA and CLR attain speedups over the base runahead system for two reasons: (1) Value prediction turns some loads that would otherwise be `INV` into useful prefeteches and (2) There is no need to roll back to the checkpoint on exiting runahead mode if the value predictions were correct. The CAVA authors evaulate both sources of speedup separately, whereas the CLR authors do not. We can gain some insight from the CAVA numbers, but we should first test agreement on the speedup delivered by basic runahead. Table 8 show the speedups measured by each author on SPECint and SPECfp using basic runahead execution in conjunction with an aggressive hardware prefetcher.

From the table, we see that the CAVA work finds significantly different speedups for runahead, so we should be careful not to compare their numbers directly to the other works. CAVA finds that adding value prediction but never eliminating rollback—a scheme called *runahead with value*

	Mutlu [20]	Ceze [3]	Kirman [10]
SPECint	13%	7%	15%
SPECfp	35%	18%	30%

Figure 8. Speedups for basic runahead execution given by three authors in independent experiments.

prediction—improves SPECint performance by 13% and SPECfp by 19% over the base superscalar. Eliminating rollback in the event of a correct value prediction—the *rollback avoidance* scheme—gives a speedup of 14% on SPECint and 33% on SPECfp over a base superscalar. Rollback avoidance is therefore much more important for SPECfp than for SPECint. For SPECint, the main effect of value prediction is to turn `INV` loads into useful prefetches.

4.3. Branch Predictability

The unpredictability of branches is a severe performance limiter in both large-window and runahead techniques. According to [4], instructions after mispredicted branches tend not to prefetch useful data. Actually, this is a point of contention, and [13] claims that wrong-path accesses can account for up to 10% of runahead's speedup on SPECint. In any case, it is best to maximize the number of correct-path prefetches.

With a typical superscalar branch predictor, [15] gives the average distance to a mispredicted `INV` branch as as 712 instructions for SPECint and 1105 instructions for SPECfp. The misprediction rate for SPECint is limiting because on average 1240 instructions can be executed per 1000-cycle runahead period. A large-window processor will encounter the same problem; more than 40 percent of the window will be squashed when a missing load returns. The fraction of useful work in the window drops steadily as memory latency increases further.

If a perfect branch predictor is assumed, runahead MLP increases by 42% over a baseline runahead processor on a collection of enterprise benchmarks (SPECweb, SPECjbb, and an unspecified database benchmark) assuming a 1000 cycle memory latency [4]. The CFP paper found that assuming perfect branch prediction gave CFP an additional 23% to 32% speedup. Clearly, it is worthwhile to incorporate better branch predictors in these designs even if they have very long latencies (perhaps tens or hundreds of cycles) [4]. Branch predictor research is not dead yet!

4.4. Inefficiency of Runahead Execution

With runahead execution, some instructions will be executed twice and others will be executed unnecessarily. As result, a processor with runahead execution consume more energy than a processor without it. Mutlu [14] identified short and overlapping runahead executions as two important causes of inefficiency in a runahead processor.

Exiting from runahead mode is expensive. The pipeline is flushed and the checkpointed register file must be restored. Therefore, if the processor stays in runahead mode for short periods of time, efficiency suffers. To avoid many short runahead periods, Mutlu [14] suggests keeping track of the number of cycles each L2 miss spends waiting for data from memory. If the processor is only allowed to enter runahead when this count crosses a threshold (rather than when the load reaches the head of the ROB), the number of dynamic runahead instructions can be reduced by 40% with very minimal IPC impact and substantial power savings.

Overlapping runahead periods are another source of inefficiency. Two runahead periods overlap if some of the instructions executed during the first period are re-executed during the second. To avoid this inefficiency, Mutlu [14] proposed beginning a new runahead interval only when it will not overlap a previous runahead interval. To achieve this, the processor counts the number n of instructions pseudo-retired during runahead mode and saves that value in a register on return to normal mode. In normal mode, it also records the number of instructions i fetched since the last runahead period. When an L2 miss reaches the head of the ROB in normal mode, the processor enters runahead mode only if $i > n$. This scheme also has minimal IPC impact but reduces the number of runahead instructions by 37%.

5. Conclusions

We have discussed three different techniques for enhancing MLP: prefetch threads, large instruction windows, and runahead execution. Of these, large instruction windows and runahead execution provide the most speedup on the widest range of applications. It is not yet clear whether the performance advantages of large window processors are worth the additional complexity or whether the comparable but slightly lower speedups offered by runahead processors at lower complexity will prove more attractive.

Even within the realm of runahead processors, there are important and contentious questions left unanswered. What should be the role of value prediction? Are full-blown rollback-avoidance schemes such as CLR and CAVA justified, or does runahead with value prediction provide the optimal performance-complexity tradeoff? It is too early to answer these questions, but we have attempted to provide and compare some of the early results.

References

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

[2] R. Barnes, E. Nystrom, J. Sias, S. Patel, N. Navarro, and W.-M. Hwu. Beating in-order stalls with 'flea-flicker' two-pass pipelining. In *MICRO-36*, 2003.

[3] L. Ceze, K. Strauss, J. Tuck, J. Renau, and J. Torrellas. CAVA: Hiding L2 misses with checkpoint-assisted value prediction. In *Computer Architecture Letters*, March 2005.

[4] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *ISCA*, 2004.

[5] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. F. Lee, D. Lavery, and J. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *ISCA*, 2001.

[6] J. Dundas. Improving processor performance by dynamically preprocessing the instruction stream, 1998.

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

[8] A. Glew. MLP yes! ILP no!, 1998.

[9] D. Kim and D. Yeung. Design and evaluation of compiler algorithms for pre-execution. In *ASPLOS*, 2002.

[10] N. Kirman, M. Kirman, M. Chaudhuri, and J. Martínez. Checkpointed early load retirement. In *HPCA*, 2005.

[11] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. A large, fast instruction window for tolerating cache misses. In *ISCA*, 2002.

[12] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *ASPLOS*, pages 138–147, 1996.

[13] O. Mutlu, H. Kim, D. Armstrong, and Y. Patt. An analysis of the performance impact of wrong-path memory references on out-of-order and runahead execution processors, 2005.

[14] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *ISCA*, 2005.

[15] O. Mutlu, H. Kim, J. Stark, and Y. Patt. On reusing the results of pre-executed instructions in a runahead execution processor. In *Computer Architecture Letters*, February 2005.

[16] O. Mutlu, J. Stark, and C. Wilkerson. Apparatus for memory communication during runahead execution, 2002.

[17] K. Olukotum, B. A. Nayfeh, L. Hammond, and K. Chang. The case for a single-chip multiprocessor. In *ASPLOS*, 1996.

[18] J. E. Smith and T. Karkhanis. A day in the life of a data cache miss. In *Workshop on Memory Performance Issues*, 2002.

[19] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual flow pipelines. In *ASPLOS*, 2004.

[20] C. Wilkerson, J. Stark, O. Mutlu, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows. In *HPCA*, 2003.

[21] H. Zhou and T. M. Conte. Enhancing memory level parallelism via recovery-free value prediction. In *ICS*, 2003.

[22] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *ISCA*, 2001.