"Flea-flicker"* Multipass Pipelining: An Alternative to the High-Power Out-of-Order Offense

Ronald D. Barnes†
Department of Electrical and Computer Engineering
George Mason University
Fairfax, VA 22030
rbarnes1@gmu.edu

Shane Ryoo   Wen-mei W. Hwu
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{sryoo, hwu}@crhc.uiuc.edu

†Work completed while at the University of Illinois at Urbana-Champaign.

Abstract

As microprocessor designs become increasingly power- and complexity-conscious, future microarchitectures must decrease their reliance on expensive dynamic scheduling structures. While compilers have generally proven adept at planning useful static instruction-level parallelism, relying solely on the compiler's instruction execution arrangement performs poorly when cache misses occur, because variable latency is not well tolerated. This paper proposes a new microarchitectural model, multipass pipelining*, that exploits meticulous compile-time scheduling on simple in-order hardware while achieving excellent cache miss tolerance through persistent advance preexecution beyond otherwise stalled instructions. The pipeline systematically makes multiple passes through instructions that follow a stalled instruction. Each pass increases the speed and energy efficiency of the subsequent ones by preserving computed results. The concept of multiple passes and successive improvement of efficiency across passes in a single pipeline distinguishes multipass pipelining from other runahead schemes. Simulation results show that the multipass technique achieves 77% of the cycle reduction of aggressive out-of-order execution relative to in-order execution. In addition, microarchitectural-level power simulation indicates that benefits of multipass are achieved at a fraction of the power overhead of full dynamic scheduling.*

1 Introduction

Out-of-order execution is a common microarchitectural strategy that allows the processor to determine how to efficiently order instruction execution. Under this model, the cost of long latency operations can be hidden by the concurrent execution of other instructions. Furthermore, since this selection is dynamic, the ordering of instruction execution can adapt to run-time conditions. Primarily because of this ability to adapt to run-time events, in particular data-cache misses, out-of-order execution is used in the majority of contemporary high-performance microprocessors [12, 13, 14].

However, the out-of-order execution mechanisms replicate, at great expense, much work which can be done effectively at compile time. Aggressive register renaming eliminates output- and anti-dependences that restrict the motion of instructions. This duplicates much of the effort of compile-time register allocation. Dynamic scheduling relies on complex scheduling queues and large instruction windows to find ready instructions, and, in choosing the order of instruction execution, repeats the work of the compile-time scheduler. These mechanisms incur significant power consumption and add instruction pipeline complexity.

A static, in-order execution model avoids this expense by executing strictly according to the compiler's specified plan of execution. While compilers can be successful at planning useful static instruction-level parallelism (ILP) for in-order microarchitectures, the efficient accommodation of unanticipable latencies, like those of load instructions, remains a vexing problem [20].

This paper introduces *multipass pipelining*, a microarchitectural technique that exploits compile-time scheduling while providing for *persistent*, advance execution of instructions otherwise blocked behind data-cache-interlocked instructions. Its performance approaches that of an ideal out-of-order execution design while incurring only a fraction of the power and complexity overhead. This is accomplished by taking multiple, carefully controlled in-order passes through instructions following what would normally be an interlock. Each pass increases the speed and energy efficiency of the subsequent passes with its valid execution results preserved in a low-complexity result buffer. These results are used to break dependences during subsequent passes, allowing instruction grouping logic to form larger

*In American football, the *flea-flicker* offense tries to catch the defense off guard with the addition of a forward pass to a lateral pass play. Defenders covering the ball carrier thus miss the tackle and, hopefully, the ensuing play. Multiple-pass pipelining utilizes two (or more) passes of pre-execution/execution to achieve its performance efficacy.

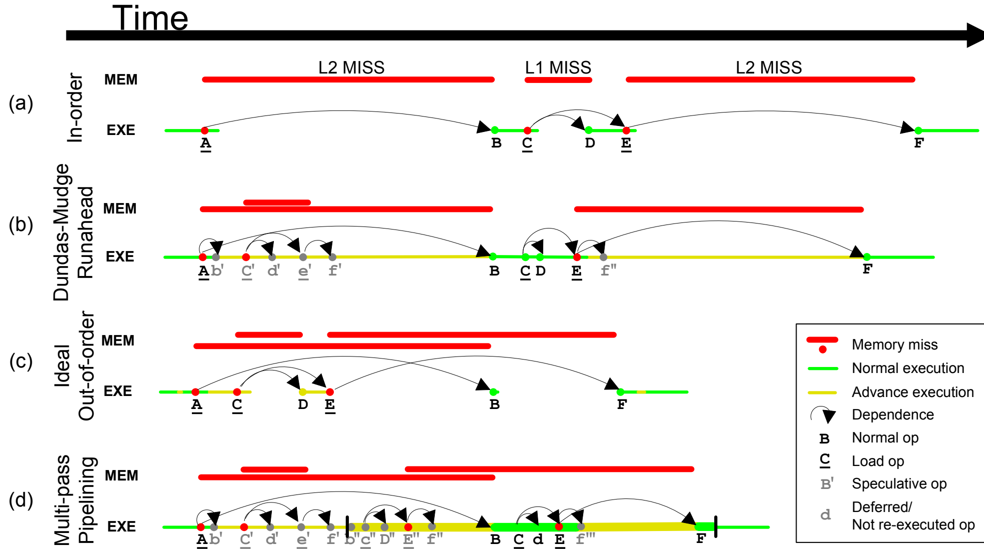

Figure 1. Execution and memory access timeline for four different instruction issue models.

issue groups without reordering instructions. During each pass, instructions with valid, persistent results from previous passes do not require further reexecution.

One important contribution of the multipass design is its ability to balance (within the same physical pipeline) the need for long-range advance execution to overlap more cache misses with the need for making multiple shorter-range advance execution passes to exploit newly-arrived, advance, shorter-latency-load miss results. This is accomplished through a mechanism that initiates the next pass when continuing the current advance execution path is unlikely to be productive. This unique ability, as we will show in the next section, helps multipass pipeline to efficiently close the performance gap between in-order execution and ideal out-of-order execution.

2 Motivation and Related Work

Figure 1 shows an example timeline repeated for several different models of execution. For each model, the execution activity is divided into actual instruction execution (**EXE**) and the handling of data cache misses caused by executing load instructions (**MEM**). In each example, the **EXE** line represents many executing instructions. A few instructions of interest are shown as lettered (`A`-`F`) points on the timeline. Instructions `A`, `C` and `E` are load instructions that miss in the data cache. Two types of misses are shown in Figure 1: relatively long misses (**L2 MISS**) and relatively short misses (**L1 MISS**). Data dependences between these instructions are shown as arrows to the dependent instruction.

Figure 1(a) demonstrates the problem that accompanies in-order processors – instructions can become artificially stalled behind consumers of load instructions which missed in the cache. When instruction `A` misses in the data cache, instructions that are independent of `A` continue to execute, causing desirable overlap between **EXE** and **MEM** activities. A stall-on-use occurs when instruction `B`, the first consumer of load `A`, is reached. For the remaining duration of `A`'s miss, the in-order processor is stalled, represented in the gap in the **EXE** timeline before instruction `B`. Similar gaps in execution are also triggered by `D` and `F`, the consumers of loads `C` and `E` that miss in cache. Performance can be improved by shrinking the **EXE** gaps (via overlap of cache misses) and/or speeding up the **EXE** segments between gaps, as we will demonstrate in the rest of Figure 1.

The original runahead preexecution approach by Dundas and Mudge [8] reduces the execution gaps by increasing the overlap between cache miss handling of independent memory loads, as illustrated in Figure 1(b). When `B` attempts to use the result of `A` before the cache miss handling completes, rather than stalling `B` and all the subsequent instructions, the runahead approach allows execution to continue in a speculative manner. This is shown as a faint continuation of the **EXE** line, marked as "advance execution," in the timeline beyond instruction `b′`.

During speculative execution, `b′` cannot compute any valid result. The execution of `b′` is feigned, and it bypasses and writes its specially marked non-result to its consumers and destination. Execution continuing past `b′` reaches independent instruction `C′` which can thus begin its memory access, overlapping its access with that of `A`. This overlap of cache miss handling of independent loads is the main source of performance improvement of the Dundas-Mudge approach and is represented in Figure 1(b) by the overlapping bold lines in the **MEM** component of the timeline.

Two limitations are evident from Figure 1(b). The first is that once an instruction's execution is skipped during runa-

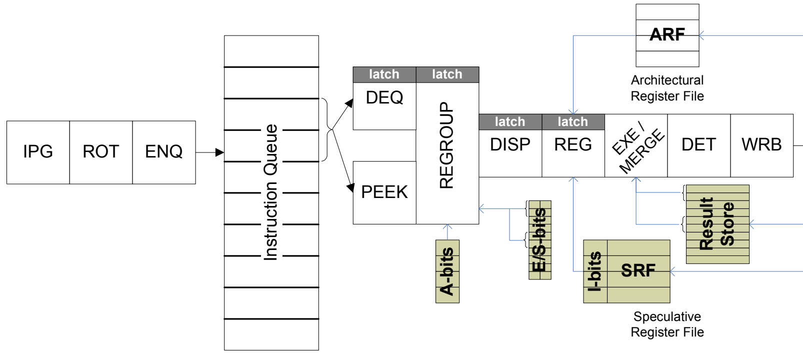

Figure 2. Integer multipass pipeline.

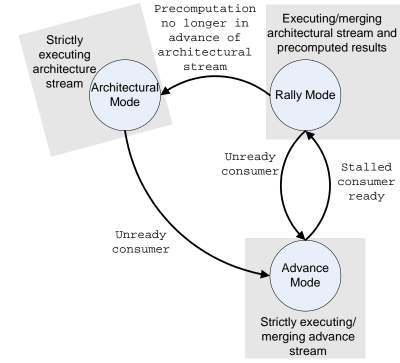

Figure 3. Three modes of multipass operation

head execution, it will not be considered again until normal execution begins again. In Figure 1(b), `e'` is skipped during runahead execution. When cache miss handling for `C'` completes, `e'` has already missed its opportunity for execution. Therefore cache miss handling for `E` cannot be overlapped with that for `A` and `C'`.

The second limitation is that none of the valid computation results from runahead execution are persistent because this is only a prefetching technique. In Figure 1(b), the pipeline still has to execute instructions `B` through `F` after the cache miss handling for `A` completes. This effectively serializes the **EXE** and **MEM** timelines and results in longer overall execution time. Furthermore, each instruction can consume execution energy multiple times. Both limitations will be addressed by the next two schemes.

Figure 1(c) shows the example timeline for an ideal out-of-order execution model. While the cache miss caused by instruction `A` is being handled, the wake up logic in the dynamic instruction scheduling mechanism allows execution of all subsequent instructions in the program's instruction stream as their operands become ready. In Figure 1(c), this allows the overlap of `C`'s miss with that of `A`. Unlike Dundas-Mudge runahead, the execution of `E` occurs immediately after the miss handling for C completes. This leads to a major benefit: the miss handling for `E` is now overlapped with that for `A`, addressing the first limitation of the Dundas-Mudge approach. Another benefit over the Dundas-Mudge approach is that instructions `C` through `E` do not need to be reexecuted after the cache miss handling for `A` completes. This can save substantial execution time and energy after a long-latency cache miss.

A large instruction window equipped with potentially very large scheduling tables, reorder buffers, and load-store queues are needed to achieve the benefits of the ideal out-of-order execution model. These benefits come at great power and complexity cost. Because of practical limitations, contemporary out-of-order processors realize only a fraction of the potential benefit. Mutlu et al. present a practical approach to tolerating long cache miss latencies by adding runahead support into a contemporary out-of-order pipeline with a modestly sized instruction window [16]. The prefetching effect of this kind of runahead execution captures similar overlap of cache-missing accesses as ideal out-of-order execution [5]. Instead, we propose multipass pipelining, with a set of enhancements to the Dundas-Mudge approach, to allow an in-order pipeline to achieve the majority of the benefits of an ideal out-of-order pipeline.

Figure 1(d) shows the execution timeline of multipass pipelining. The behavior of the multipass approach has two important differences from the Dundas-Mudge runahead model. First, the pipeline can make multiple passes through the instructions subsequent to the consumer of a missing load. During advance execution, as instructions are suppressed because of unavailable source operands, the speculative state may become so contaminated that continued advance execution is fruitless. Rather than wasting execution effort further down the instruction stream, the advance execution is restarted at the consumer instruction that triggered the advance execution. In Figure 1(d), the multipass pipeline restarts the advance execution at `b''` shortly after instruction `f'`. During the second pass, the short cache miss handling for `C'` has completed. Therefore, `E''` in the second pass can now trigger its cache miss handling before the miss handling for `A` completes, addressing the first limitation of the Dundas and Mudge runahead model. The mechanism for triggering the restart of the advance execution will be discussed in the following section.

The multipass pipeline restarts the runahead execution at the original consumer instruction `B`, although it is not yet ready for execution. This shows an important difference between the multipass pipeline and an ideal out-of-order pipeline, where the wake-up logic would have used the `C` load result to select `D` and `E` for execution. Such accurate wake-up logic would be costly in power. Instead, the multipass pipelining model simply restarts the advance

execution, hoping to find additional instructions, such as `D''` and `E''`, whose input operands have become ready between pass one and pass two.

The second difference is that the multipass pipeline preserves valid execution results during advance execution and uses them to reduce power consumption and speed up execution during subsequent execution. When advance restart occurs, the preserved results are used to avoid executing speculative instructions like `c''` again. This also allows the second pass in Figure 1(d) to reach `E''` more rapidly, since the instructions with preserved execution results no longer have data flow dependence on other instructions. When the processor returns to normal execution, the preserved results for `C` and `d` are used to speed up the processing of these instructions while saving energy. This addresses the second limitation of the Dundas-Mudge runahead model. A previous approach, flea-flicker two-pass pipelining [2], also reused preexecution results, but required replication of the execution pipelines and did not support the restart of advance execution.

3 Microarchitecture

The multipass pipelining scheme is designed to allow the productive processing of independent instructions during the memory stall cycles left exposed in traditional in-order pipelines. We will illustrate its design by adapting a contemporary in-order pipeline design, that of the Itanium 2 [15], as shown in Figure 2. Multipass additions to the pipeline are darkened.

In an in-order processor, fetched instructions are often buffered to decouple instruction fetch from execution. One implementation of the Itanium 2 processor has a buffer that holds up to 24 instructions or at least four cycles worth of instructions. The multipass pipeline extends such an instruction buffer in size and purpose while maintaining its simplicity as a FIFO buffer. To accommodate the longer delay of a larger buffer, two new stages that enqueue (**ENQ**) and dequeue (**DEQ**) (or peek at (**PEEK**)) instructions in the buffer are shown in Figure 2. A third stage is added to perform instruction regrouping (**REGROUP**), as described in Section 3.2.

3.1 Overview of multipass operation

Multipass pipelining performs persistent, advance execution when an in-order processor would be otherwise stalled. Because both normal and advance execution occur on the same physical pipeline at different times, this pipeline operates in different modes, shown in Figure 3.

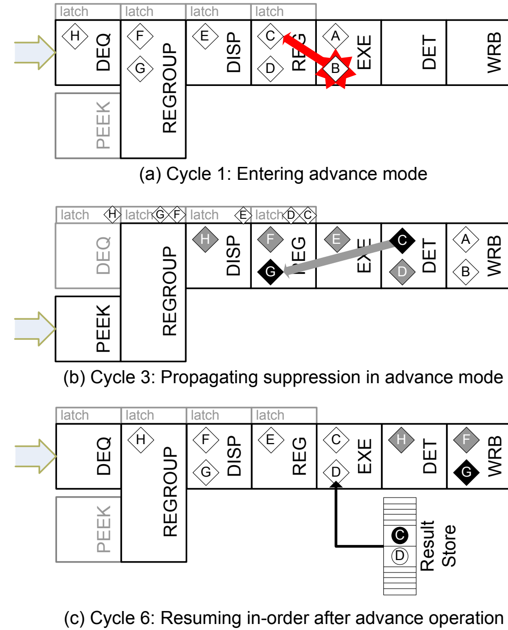

Figure 4. Multipass pipelining operation.

3.1.1 Architectural execution

Initially, the pipeline enters the *architectural* mode when the execution of a program starts. In the absence of runtime stalls, instructions are released from the instruction buffer using the **DEQ** pointer. The release and execution of these instructions resembles that of conventional in-order execution pipelines. During architectural mode, the structures specific to multipass pipelining are unused and can be clock gated for power efficiency.

3.1.2 Advance execution

Multipass advance preexecution begins with the failure of an instruction to receive a valid operand. For example, in Figure 4(a), load `B` misses in cache, causing dependence checking logic in the **REG** stage to detect an unready operand for instruction `C`. At this point, the pipeline enters *advance* mode. All in-flight instructions from the triggering instruction back to the instruction queue (`C` to `H`) are latched at their respective stages, for reasons discussed later in the section. The **DEQ** pointer is also preserved and subsequent instructions are released from the instruction buffer using a **PEEK** pointer. These proceed through the pipeline as advance instructions.

During advance mode, any instructions failing to receive valid input operands are simply suppressed. An *invalid* (I) tag is attached to their output values to indicate that these instructions were deferred. This in turn suppresses consumers of the suppressed instructions' result. The multipass pipeline in advance mode selectively executes only the advance stream instructions that receive valid input data.

Advance stream instructions are not allowed to write their results into the architectural register file (ARF). Instead, their results are redirected to the speculative register file (SRF), which stores the speculative state for the current pass of advance preexecution. When the pipeline enters advance mode, SRF does not contain any valid information; advance stream instructions initially access ARF for their input operands. As advance stream instructions write into SRF, the consumers of their results need to be redirected to this file for input operands. This redirection is realized with a bit vector, shown as A in Figure 2. Each *advance* bit (A-bit) indicates that future accesses to its associated register entry should be redirected to SRF.

During architectural mode, the A-bits are clear, and all instructions read operands from the ARF. In advance mode, each instruction sets the A-bits associated with its destination register(s), directing their subsequent consumers to fetch from SRF. Each SRF entry contains an I-bit that marks invalid values written by suppressed advance stream instructions. Advance instructions reading a register with a set I-bit are suppressed. The logic for bypassing between in-flight advance stream instructions is detailed in Section 3.4. Advance instructions are also not allowed to change the memory state; Section 3.6 explains a simple advance store cache to enforce memory dependences and forward memory values.

An important feature of the advance mode is that results of the correctly executed advance stream instructions are preserved in an result store (RS). The RS is written in addition to the SRF by advance instructions. There is a one-to-one correspondence between instruction buffer and RS entries. The RS entries corresponding to suppressed advance stream instructions are simply marked as *empty* with an E-bit. In Figure 2, the E-bit vector with entries corresponding to the RS is read in the **REG** stage.

During advance mode, the advance restart mechanism, explained in Section 3.3, determines if advance mode execution should be returned to the instruction that originally triggered the current advance mode execution, e.g. instruction `C` in Figure 4(b). At this point, all A-bits are cleared, effectively clearing the SRF.

During advance mode restart, the contents of the instruction queue and the result store (RS) remain preserved. The RS contents are used to speed up the processing of advance instructions if these instructions have been successfully executed in a previous pass of advance mode. The E-bits are used by the regrouping logic to determine the instructions that no longer have flow- or anti-dependences due to the availability of their result value. The reduced dependences allow the regrouping logic to form larger issue groups without reordering these instructions, thus allowing the pipeline to process the same instruction stream faster. The result store has two other benefits: first, the pipeline does not have to spend the energy to execute an instruction whose results are available from prior advance-mode execution; second, long-latency instructions, such as multiply instructions, are effectively converted into single cycle instructions with this feature, further reducing potential stalls in rally mode.

3.1.3 Rally Mode

During advance mode, the availability of the input operand of the instruction that triggered advance mode is monitored. When its input operand becomes available for bypass in the **REG** stage, the pipeline switches to *rally* mode wherein architecture-stream instructions resume execution. The latched instructions are unlatched and displace the advance-mode instructions in their respective stages. Unlike advance restart, the contents of the latches are not maintained. As was the case of a restart of the advance mode, the rally mode uses RS contents and the A-bits to speed up the execution of architecture-stream instructions that have already been correctly preexecuted in advance mode.

If any architecture-stream instruction receives an invalid operand bypass value at the **REG** stage, the pipeline switches to advance mode again. Alternatively, if the **DEQ** pointer reaches the farthest point of the preserved **PEEK** pointer while in the rally mode, then the architecture stream has caught up with the farthest point of preexecution. This indicates that there are no longer any instructions deferred on pending cache misses. The pipeline can now switch back to architectural mode, shutting off the multipass-specific structures.

In Figure 4(c), architectural execution has resumed with the in-order dequeuing of instructions. Instructions that were correctly preexecuted simply read their result from the RS rather than reexecuting. When earlier preexecution consisted of cache misses (or other long-latency operations) as in the example in Figure 1, architectural execution is likely accelerated by the elimination of future stalls.

3.2 Instruction regrouping

Because of the persistent execution performed during advance mode, much of rally mode execution consists of merely merging precomputed instruction results into the processor state. Because the results of precomputed instructions are not recomputed (with the exception of data-speculative loads as detailed in Section 3.6), such instructions can be considered to no longer be dependent on the original producers of their source operands. This elimination of input dependences permits an optimization called *issue regrouping*. New issue groups can be formed without changing the compiler-specified instruction *order*.

Instruction regrouping is done by checking dependences on an instruction-by-instruction basis in the **REGROUP** stage (as would be done in a dependency check stage in a non-EPIC in-order processor). Preexecuted instructions,

marked with their corresponding E-bits, are independent of all other instructions, thus allowing a dynamic schedule compaction beyond what was possible at compile time.

3.3 Advance execution restart

As advance execution proceeds, often a point is reached where little fruitful forward advance execution can be performed because the vast majority of subsequent instructions are dependent upon cache-missing loads or deferred instructions. At the same time, instructions which have previously been deferred because of an unready operand may now represent an opportunity for productive preexecution. The general wakeup mechanisms of out-of-order execution allow such instructions to execute as soon as their operands are ready. Multipass pipelining achieves the same benefit by relying on the systematic restart of advance execution.

A very simplified notion of critical instructions [19, 21] is used to control advance restart. Restart may be desirable if a deferred instruction will cause the vast majority of subsequent preexecution to be deferred. For the results presented in this work, RESTART instructions are explicitly inserted by the compiler to direct advance restart.[1] During compile time, strongly connected components (SCCs) of the data-flow graph are found: these components represent loop-carried data flow. If an SCC precedes a much larger number of multiple-cycle or variable-latency (such as load) instructions than the SCC succeeds in the data-flow graph, the loads in the SCC are considered critical. A RESTART is inserted after every load in the SCC, consuming the load's destination. When the RESTART 's operand is unready, advance restart occurs, otherwise the instruction has no effect. The A-bit vector and advance store cache are cleared, latched instructions in the **DEQ**, **DISP**, **REGROUP** and **REG** stages are unlatched (but preserved for future restart).[2]

3.4 Disseminating advance values

The multipass pipelining model must accommodate architectural and advance streams in a single pipeline without co-mingling their values in an undesired fashion. This involves preventing spurious bypasses and respecting certain output dependences. Bypasses between advance and architectural mode instructions are easily prevented through the addition of the A-bit to each register identifier in the bypass network, indicating whether an advance or architectural instruction generated the value being bypassed. Advance instructions set the A-bit of their destinations in the **EXE** stage denoting that advance preexecution supersedes the value stored in the ARF for that register. The A-bit of each instruction operand is read during advance mode in the **REGROUP** stage, dictating to an instruction which register file to later read in the **REG** stage. Some advance instructions may read stale values from the ARF if the producer of their operand has yet to write the A-bit vector, but the appropriate advance value will be provided later via the bypass network. Advance instructions accept the bypass of the most recently executed instruction; architectural instructions ignore bypasses marked with the A-bit.

3.5 Handling write-after-write dependences

In an EPIC implementation such as the Itanium 2, all instructions are issued strictly in-order, but variable-latency instructions might complete out-of-order, since a shorter-latency writer might follow a longer-latency writer of the same operand. Out-of-order instruction completions cannot be allowed to cause inconsistent register state. Since EPIC processors do not dynamically rename register operands, variable-cycle latency instructions (in particular loads) are scoreboarded to force output dependent instructions to stall.

Similarly, the architectural stream of execution stalls when write-after-write (WAW) dependencies present themselves. However, an alternate approach is preferred for the execution of advance instructions. Dynamic WAW dependencies are reached frequently in loops, as dynamic instances of the same static instruction are obviously output dependent. Additionally, when a WAW is reached in advance execution, all consumers of the first write have already been processed (and deferred) so there is no reason to stall on these writes. The simplest alternative approach, proposed here, is for none of the advance load instructions that miss in the first level cache to write back to the speculative register file and avoid WAW concerns entirely. These loads will eventually write their results to the RS, but all consumers of missing loads will be deferred until a subsequent pass. Alternative solutions (requiring more complexity) would suppress the register file write back of loads only once a WAW occurs.

3.6 Handling advance memory instructions

A multipass-pipelined system maintains an underlying in-order execution model. Advance-stream instructions, since they are processed out of program order from the architecture stream, are speculative and their processing must not directly affect architectural state. The purely speculative processing of non-memory instructions is handled simply with the addition of the SRF. Memory instructions require additional treatment.

All load and store instructions are allocated entries in an address table at the time of instruction dispersal. Pre-

[1] A hardware mechanism could also have been used to detect these situations.

[2] Alternatively a microarchitectural mechanism could be used to redirect PEEK to the initial advance instruction early, so that the instruction arrives at the REG stage at the same time as its input.

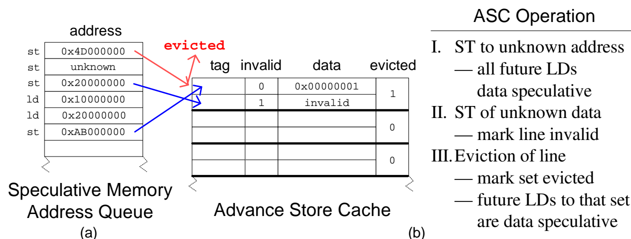

Figure 5. Multipass memory consistency structures.

executing (but not deferred) memory instructions enter the address of their access in the speculative memory address queue (SMAQ), shown in Figure 5. The SMAQ is used by advance stream memory instructions to avoid rereading their address operands in rally mode. An advance store's data operand is also preserved in the RS and reused in a similar manner. Though they do not reread their input operands, preexecuted stores and dynamically data-speculative loads perform memory accesses in rally mode as described later in this section. Therefore, the SMAQ and RS together allow instruction regrouping to place preexecuted memory instructions in the same execution cycle as their address or data calculation instructions.

Traditionally, store buffers are used to support forwarding data that is not yet visible in the data cache from store instructions to load instructions. However, in-order processors have little need for large store buffers. To support a much wider window of in-flight stores and loads in a multipass pipeline, the advance store cache (ASC) is a low-associativity cache structure to nominally forward data during advance mode. Figure 5(b) shows the ASC and outlines its function.

At the beginning of each pass of advance execution, the ASC is cleared. Load instructions access the main cache hierarchy for their data. As advance stores execute, they deposit their data into the ASC (not to the traditional cache). Subsequent advance loads access both traditional cache and ASC, with hits in the ASC overriding those from the traditional cache, As long as advance store values can be forwarded through the ASC, a consistent memory interface is maintained. For example, if a store has an invalid data operand, the result of a load to the same location is also invalid. However, advance stores may be deferred due to an unavailable target address. Out-of-order processors [14] tend to use content-addressable load-store queues for detecting when a load is dynamically reordered with a conflicting store. This approach could be used by multipass pipelining, but its unnecessary hardware complexity limits the reorderable window of instructions. Replacement in the ASC and the ASC's low associativity allow it to support a large window of instructions while communicating to subsequent loads values that are either correct, invalid, or data speculative (because of replacement). Exploiting the fact that advance-mode instructions will be processed again after they are dequeued in future passes, multipass pipelining takes a value-based approach [17] to preserving memory consistency. This approach is made possible by the fact that each instruction will be processed in-order in rally mode exactly once.

If a store instruction is deferred because of an unknown address operand, all future load instructions (and their dependents) are data speculative. Similarly, advance load instructions that miss from ASC sets that have undergone replacement are treated as data speculative. When such load instructions are processed in advance mode, their results are marked data-speculative with a set S-bit corresponding to that instruction's RS entry. When these data speculative instructions are reprocessed in rally mode, they will reperform their memory accesses, using their addresses from the SMAQ. If the value loaded is not the same as the value that was loaded during advance mode, a pipeline flush is performed.

4 Evaluating the cost of multipass execution

Justifying the overhead of the multipass pipelining requires not only examination of the performance potential demonstrated in the next section, but also a detailed evaluation of the additional hardware complexity and power. Because the more conventional approach to tolerating dynamic memory latency is through out-of-order execution, multipass pipelining will be compared to the complexity of an out-of-order design. The relative expense of analogous structures and estimations for structures unique to the particular designs can provide insight into the relative complexity of these implementations.

A comparison of the multipass and out-of-order hardware structures was performed using microarchitectural power models adapted from Wattch [3]. Wattch-supplied technology parameter estimates for a 100nm process were chosen to approximate device characteristics of a contemporary high-performance microprocessor using a V_{dd} of 1.2V and a frequency of 2GHz. In the evaluated architecture, 128 integer, 128 floating point, and 64 predicate registers are visible to the instruction set. Data and memory addresses are 32 bits wide and data is associated with an additional NaT bit [15] for compiler speculation support. Decoded instructions are 41 bits wide and 6 instructions can be issued per cycle. Table 1 shows the ratio of the power of structures specific to out-of-order execution in comparison to multipass structures which serve similar purposes: a ratio greater than 1 indicates higher out-of-order power. The peak power ratio assumes maximum switching activity. The average power ratio is based on simulated results and Wattch's linear clock gating model: it was measured by

Table 1. Power Ratios of Out-of-Order to Multipass Structures in a 100nm process.

Out-of-Order Structures	Multipass Structures	Peak Power Ratio	Average Power Ratio
Combined Architectural & Renamed Register File (512 registers, 12R/8W ports) Register Alias Table (array: 256 entries, 9 bits, 12R/6W ports)	Architectural & Speculative Register Files (each 256 registers, 12R/8W ports) Result Store (2-banked array: 256 entries, 1 wide-read & 1 wide-write & 2 single-write ports)	0.99[3]	1.20
Instruction Wakeup (wired-OR resource dependence matrix: 128 entries, 329 bits) Instruction Issue (128 entries, 19 bits, 6R/6W ports)	Instruction Queue (2-banked array: 256 entries, 1 wide-read & 1 wide-write port)	10.28	7.15
Load Buffer (CAM: 48 entries, 2R/2W ports) Store Buffer (CAM: 32 entries, 2R/2W ports)	Speculative Memory Address Queue (SMAQ) (2-banked array, 128 entries, 2R/2W ports) Advance Store Cache (ASC) (2-way set associative cache, 64 entries, 2R/2W ports)	3.21	9.79

incorporating the relevant Wattch component models into the cycle-by-cycle simulator used for performance results.

The Wattch component power models consist primarily of array components: decoders, wordlines, bitlines, and senseamps. For these structures, power is expected to scale nearly linearly with the number of ports. An additional effect is encountered due to the increase in cell size because of the additional wordlines and bitlines needed for access. Since content-addressable memories (CAMs) must read out their entire contents and match them, they are far more costly in power than indexed arrays. The primary sources of multipass power savings are the avoidance of CAMs and the reduction in the number of ports due to always-sequential execution. Table 1 is only meant to illustrate the degree of disparity between out-of-order and multipass structures, and not to represent the power consumption of any physical implementation.

4.1 The cost of out-of-order execution

The structures central to out-of-order design can be very expensive. For example, in the Alpha 21264 processor, the out-of-order logic consumes 18% of the total power–almost as much as all of its integer and floating point execution units combined [11]. While aggressive out-of-order implementations are desired to find more independent instructions and thus maximize the tolerance of cache miss latency, power concerns are driving more conservative implementations of out-of-order processors [10]. In order to conservatively show the favorable power advantages of multipass pipelining, we chose structures with lower power cost when possible and neglect several potentially expensive aspects of out-of-order execution. A model similar to that of the Pentium III [12] is assumed, but any register copy cost that would be incurred in the reorder buffer is ignored. We have also omitted the renaming hardware for predicate registers: because the architecture is capable of writing 12 predicate registers per cycle, conventional out-of-order mechanisms are likely to be impractical.

Decomposing the costs of out-of-order execution, a typical out-of-order implementation consists of three processes: register renaming, dynamic scheduling, and instruction reordering for retirement. To eliminate false dependences created because the reuse of the same architectural register names, register renaming is needed to make dynamic scheduling effective. With dynamic scheduling, the processor itself decides the order of instruction execution, issuing instructions when their data-flow dependences are met. Lastly, to insure that instruction execution affects architectural state in a way consistent with the original program, the results of instruction execution are buffered for incorporation in program order. Conventional implementation of these functionalities requires CAMs and other many-ported structures to implement, which impacts cycle time and incurs a large power cost [18].

[3]The 0.99 power ratio requires double the register file accesses that the architecture can actually incur; it is an intentionally conservative estimate. Under architectural constraints the peak power would be 1.92.

4.2 Overhead of multipass pipelining

Multipass pipelining requires a second register file for advance execution. However, instructions issuing in the single physical pipeline read either the advance or the architectural register file for each of their operands. Thus, ports can be shared for both the advance and architectural register storage. In a recently announced Itanium 2 processor, a similar register file with storage for two register values for every architectural register has been implemented to support simultaneous multithreading, with only a 15% increase in area [9]. For the power evaluation we conservatively assume two separate register files of 256 registers each, which is the reason multipass peak power is slightly worse than out-of-order (with its equal capacity monolithic file) for the register/data structures.

The result store needs to only support a single wide-read port that is as wide as the issue width. Since RS writes are not always aligned on specific 6-instruction boundaries, individually-qualified wordlines are necessary for each result (bitlines can be shared due to sequentiality). Because some results return with non-unit latency, the result store must also support individual write ports for these results.

The multipass instruction queue (IQ) is a large but simple queue. It requires only one wide-read port for dequeuing and peeking and one wide-write port for enqueueing, and is significantly simpler than the general scheduling ta-

ble required by dynamic scheduling. The scheduling table supports the selection and reading of any general instruction in the table, and needs a port for every instruction issued simultaneously in a given cycle. A CAM-based scheduler was also tried and found to have higher power consumption. Additionally, while instruction dequeuing (or peeking) occurs strictly in-order from the queue, conventional scheduling tables perform tag comparisons between every register destination generated by execution and every register source of instructions awaiting issue.

The overhead of insuring the proper semantic ordering of memory loads and stores is greatly reduced in the implementations proposed for multipass pipelining. Instead of the load/store ordering queue used in out-of-order execution, multipass pipelining allows the reordering of loads and stores by verifying that the value loaded by data-speculative loads is correct. The hardware structures required for this are the SMAQ and ASC, which have lower peak power compared to the out-of-order CAMs despite having more entries. The results in Section 5 demonstrate that performance stalls are not significantly impacted by the pipeline flushes caused by the maintenance of semantic memory ordering since conflicts between the loads and stores were rarely observed in our models.

5 Experimental Results

A number of experiments were conducted to test the effectiveness of multipass pipelining. While the technique is applicable across in-order microarchitectures, an EPIC platform based loosely on the Itanium 2 architecture was chosen for these studies.

5.1 Evaluation Setup

Twelve C-language benchmarks were selected from SPEC CPU2000 to test the performance of multipass pipelining. These represent a wide variety of application types; the remaining CINT2000 benchmarks were excluded due to compilation issues. Each application was compiled through the OpenIMPACT compiler [23] using the SPEC-distributed training inputs to generate basic block profile information. Interprocedural points-to analysis was used to determine independence of load and store instructions, enabling aggressive code reordering during optimizations. Optimizations performed include aggressive inlining, hyperblock formation, control speculation, modulo scheduling, and acyclic intra-hyperblock instruction scheduling. OpenIMPACT does not perform optimizations specifically targeting floating point applications. Results in this work reflect rigorously sampled [25], complete runs of SPEC reference inputs.

To evaluate the multipass pipelining paradigm, an in-order model, a multipass model and an idealized out-of-order simulation model were developed. Table 2 shows the relevant machine parameters derived from the Intel Itanium 2 design. This contemporary cache hierarchy was chosen to model an achievable near-term design; forward looking cache parameters are also examined in Section 5.3. Results in this work assumed a model using 32-bit pointers; the use of 64-bit pointers would increase the data footprint, increasing the number of cache misses and furnishing more opportunity for cache-miss tolerance.

Table 2. Experimental machine configuration.

Feature	Parameters
Functional Units	6-issue, Itanium 2 FU distribution
Data model	ILP32 (integer, long, and pointer are 32 bits)
L1I Cache	1 cycle, 16KB, 4-way, 64B lines
L1D Cache	1 cycle, 16KB, 4-way, 64B lines
L2 Cache	5 cycles, 256KB, 8-way, 128B lines
L3 Cache	12 cycles, 3MB, 12-way, 128B lines
Max Outstanding Misses	16
Main Memory	145 cycles
Branch Predictor	1024-entry gshare
Multipass Instruction Queue	256 entry
Out-of-Order Scheduling Window	128 entry
Out-of-Order Reorder Buffer	256 entry
Out-of-Order Scheduling and Renaming Stages	3 additional stages
Out-of-Order Predicated Renaming	ideal

The out-of-order model used for comparison with multipass pipelining was constructed to give an idealized indication of the performance opportunities from dynamically ordering instructions. Some of the performance limiting overheads of out-of-order execution mentioned in Section 4.1 were excluded from the model to demonstrate the relatively ideal performance potential from dynamic scheduling. One example is that scheduling and register file read are performed in the **REG** stage, eliminating the need for speculative wakeup of instructions as in modern implementations. Another is an ideal register renamer, which ignores the issues of register renaming in the presence of predicated code and avoids the performance cost of a realistic implementation as described in [24].

5.2 Multipass Pipelining Performance

Benchmark execution cycle counts are shown in Figure 6 for baseline inorder (**inorder**), multipass pipelining (**MP**) and out-of-order (**OOO**) models, normalized to the number of cycles in the baseline machine. Within each bar, execution cycles are attributed to four categories: *execution*, in which instructions are issuing without delay; *front-end*, stalls including branch misprediction flushes and instruction cache misses; *other*, stalls on multiplies, divides, floating-point arithmetic and other non-unit-latency instructions, and stalls on resource conflicts; and *load*, stalls on consumption of unready load results. For multipass pipelining in advance mode, cycles when no new in-

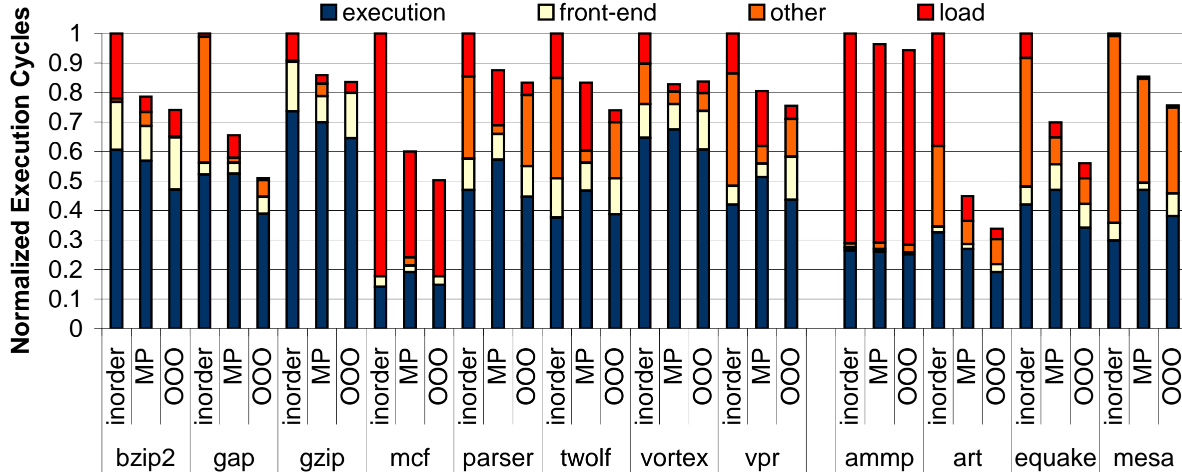

Figure 6. Normalized execution cycles; baseline (base), multipass (MP) and out-of-order (OOO).

struction executions occur (as opposed to merges or deferrals) are attributed to the unsatisfied latency that initiated advance mode. Analogously, cycles when out-of-order execution does not execute a single instruction are attributed to the cause of the stall of its oldest instruction (or as a front-end stall in the case of an empty instruction queue).

A significant number of memory stall cycles are eliminated through multipass pipelining for each benchmark. For example, *mcf*, the CINT2000 benchmark with the worst cache-miss behavior shows a 56% reduction in memory stall cycles and a 47% reduction in overall stall cycles. In some cases, such as in *bzip2*, the reduction in cache-miss stalls is partially offset by other stalls (such as those on non-unit latency instructions) that are exposed while tolerating the misses. In other benchmarks, a slight reduction in total execution cycles is achieved through preexecution of branch instructions. For example, in *twolf*, a 29% reduction in front-end stall cycles is achieved.

The average reduction in total stall cycles (both load and non-load) due to application of multipass pipelining is 49%, yielding a 1.36× average speedup. Overall, ideal out-of-order execution only achieves an additional 1.14× speedup over a multipass pipelined system from its ability to find ILP by reordering instruction executions and its more general tolerance of run-time latency.

The out-of-order model evaluated is very aggressive, so a model using decentralized scheduling tables for memory, floating point and integer instructions with 16 entries each was also examined. Using the methodology of Section 4 the relative power of these issue queues was still found to be larger, but on the same order of magnitude as that of the multipass issue queue. Because of the reduced parallelism achieved with the more quickly filled scheduling resources of this out-of-order model, multipass pipelining achieves a speedup of 1.05× over this model.

5.3 Performance with Other Cache Hierarchies

Figure 7 demonstrates the performance impact of changing the cache hierarchy from the aggressive, contemporary model shown in Table 2. High-performance in-order processors address their intolerance of cache misses through large, fast caches. As processor speeds increase, the relative speed to main memory will also likely increase. Additionally, because of power constraints, very large, low-latency caches will be increasingly difficult to implement. Three cache hierarchies are evaluated, with increasing cycle latency and decreasing cache sizes. The speedup over an in-order processor is shown for both the multipass and out-of-order model. In general, as the average latency of a memory access is increased due to the less-effective caching hierarchies, the latency tolerance effectiveness of both multipass pipelining and out-of-order execution vary across benchmarks but remain the same on average. However, the difference between multipass and out-of-order performance narrows with the more restrictive hierarchies.

5.4 Evaluating instruction regrouping and advance restart

Figure 8 shows the percent of the full multipass speedup achieved without one or the other of the key elements of multipass pipelining. Figure 8 shows that for all benchmarks except for *mcf*, instruction regrouping is responsible for a considerable amount of the speedup of the multipass approach. Advance restart is responsible for a significant component of the speedup of *bzip2*, *gap* and *mcf*, but little performance is lost by not performing instruction restart on the remaining benchmarks. These benchmarks both have fewer chained cache misses (in particular in the CFP2000 benchmarks) and fewer misses that comprise important strongly-connected components used to drive the

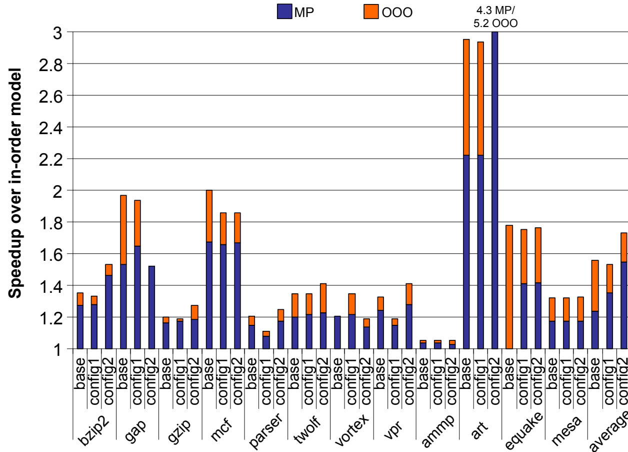

Figure 7. Speedup of multipass pipelining and out-of-order execution with varying cache sizes and latencies: base, config1 (base with 200 cycle main memory) and config2 (1 cycle 8KB L1/7 cycle 128KB L2/16 cycle 1.5MB L3/200 cycle MM)

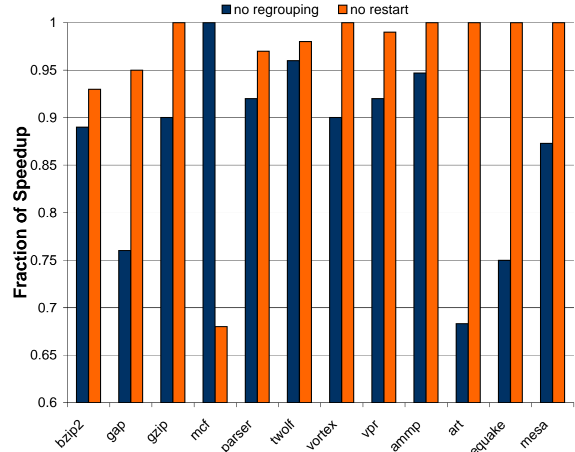

Figure 8. Relative speedups of multipass pipelining without instruction regrouping or advance restart.

advance restart described in Section 3.3. Other mechanisms for initiating advanced restart might provide benefit in applications that suffer from a mix of short and long misses, but lack the behavior identified with critical instructions. Dundas-Mudge runahead was simulated separately from results shown in Figure 6, but only reduced half as many cycles as multipass relative to in-order.

6 Additional Related Work

This work is not alone in proposing a mechanism for improving the tolerance of variable-latency instructions through preexecution of the running program. Dundas and Mudge [8] proposed an in-order preexecution implementation, and Mutlu et al. [16] presented an implementation called runahead execution that targets long-latency misses in out-of-order machines. Both approaches used preexecution purely as a prefetching technique. Preexecuted results independent of cache misses were discarded. Multu et al. found the benefit of the reuse of results in an out-of-order execution runahead implementation to be insignificant, largely because the amount of reuse in this approach was small. This runahead execution only occurs after the reorder buffer-limited instruction window was full–a small percentage of overall time.

Continual flow pipelines [22] use an approach to out-of-order execution that subsumes runahead execution. Through the use of nonblocking dynamic scheduling similar to that of the Pentium 4 [12] and reorder buffer checkpointing [1], a very large instruction window is achieved with an implementable scheduling table and register file. While continual flow pipelines achieve the large instruction window of runahead approaches while performing only persistent execution, they require the complexity of dynamic scheduling and register renaming.

In addition to runahead approaches, several thread-based approaches [4, 6, 7, 26] perform preexecution of (a subset of) application code to achieve similar benefits. However, these techniques require software thread generation, dynamic microthread formation or slice extraction for individual loads and thus do not address the more diffuse, occasional misses tolerated through multipass pipelining.

7 Conclusions

Because of the disparity between processor logic and memory speed, tolerating cache misses through dynamic scheduling has become almost an ubiquitous characteristic of modern high-performance processors. While out-of-program-order execution can tolerate variable memory-instruction latency, it adds hardware components that are problematic for power-conscious design and whose complexity limits the practical ability to reorder instructions.

Multipass pipelining tolerates long latencies (in particular unanticipated data-cache memory latency) without the overhead associated with dynamic scheduling or register renaming. Unlike most preexecution schemes, multipass pipelining provides for the persistence of valid advance execution results. Reusing these results increases efficiency, hides the latency of multiple-cycle instructions and, through a novel mechanism, accelerates in-order execution. A notion of instruction criticality further enhances the handling of miss latencies and reduces fruitless speculative execution by indicating when there is little opportunity for advance execution. Ideal out-of-order execution

achieves only a $1.14\times$ speedup over multipass pipelining with significantly more hardware complexity. When compared with a more realistic out-of-order execution implementation, multipass achieves superior performance while maintaining major power advantages.

8 Acknowledgments

The authors would like to thank John Sias, Erik Nystrom and Sanjay Patel for their contributions to the flea-flicker technique, and to additionally thank John for his assistance in critical instruction identification. We thank Matthew Merten at Intel Corporation, Hillery Hunter at IBM Research, and our anonymous reviewers for their constructive feedback. We lastly acknowledge all the members of the IMPACT research group for their feedback and assistance. This work was partially supported by the MACRO Gigascale Systems Research Center (GSRC), NSF ITR Grant 86096, and generous gift funds from Intel Corporation.

References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction windows processors. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 423–434, Dec. 2003.

[2] R. D. Barnes et al. Beating in-order stalls with "flea-flicker" two-pass pipelining. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 387–398, Nov. 2003.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[4] R. S. Chappell et al. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 186–195, July 1999.

[5] Y. Chou, B. Fahs, and S. G. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 76–87, June 2001.

[6] J. D. Collins et al. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 14–25, July 2001.

[7] M. Dubois. Fighting the memory wall with assisted execution. In *Proceedings of the First Conference on Computing Frontiers*, pages 168–180, Apr. 2004.

[8] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th Annual International Conference on Supercomputing*, pages 66–75, June 1997.

[9] E. S. Fetzer, L. Wang, and J. Jones. The multi-threaded, parity-protected 128-word register files on a dual-core Itanium-family processor. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 382–383,605, Feb. 2005.

[10] S. Gochman et al. The Intel Pentium M processor: Microarchitecture and performance. *Intel Technology Journal*, 7(2):21–36, May 2003.

[11] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power considerations in the design of the Alpha 21264 microprocessor. In *Proceedings of the 1998 Design Automation Conference*, pages 726–731, June 1998.

[12] G. Hinton et al. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal Q1*, 2001.

[13] C. N. Keltcher et al. The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, pages 66–76, March/April 2003.

[14] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19, March/April 1999.

[15] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, 23(2):44–55, March 2003.

[16] O. Mutlu et al. 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*, pages 129–140, Feb. 2003.

[17] S. Onder and R. Gupta. Dynamic memory disambiguation in the presence of of out-of-order store issuing. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 170–176, Nov. 1999.

[18] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective supersclar processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[19] A. Roth and G. S. Sohi. Speculative data-driven multithreading. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 37–49, Jan. 2001.

[20] J. W. Sias et al. Field-testing IMPACT EPIC research results in Itanium 2. In *Proceedings of the 31th Annual International Symposium on Computer Architecture*, pages 26–37, July 2004.

[21] S. Srinivasan et al. Locality vs. criticality. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 132–143, July 2001.

[22] S. T. Srinivasan et al. Continual flow pipelines. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–119, Oct. 2004.

[23] UIUC OpenIMPACT Effort. The OpenIMPACT IA-64 Compiler. (http://gelato.uiuc.edu/).

[24] P. H. Wang et al. Register renaming and scheduling for dynamic execution of predicated code. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 15–25, Jan. 2001.

[25] R. E. Wunderlich et al. SMARTS: Accelerating microarchitectural simulation via rigorous statistical sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 84–95, June 2003.

[26] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 2–13, July 2001.