Beating In-Order Stalls with "Flea-Flicker" Two-Pass Pipelining

Ronald D. Barnes, *Member*, *IEEE*, John W. Sias, *Student Member*, *IEEE*, Erik M. Nystrom, *Student Member*, *IEEE*, Sanjay J. Patel, *Member*, *IEEE*, Jose (Nacho) Navarro, *Member*, *IEEE*, and Wen-mei W. Hwu, *Fellow*, *IEEE*

Abstract—While compilers have generally proven adept at planning useful static instruction-level parallelism for in-order microarchitectures, the efficient accommodation of unanticipable latencies, like those of load instructions, remains a vexing problem. Traditional out-of-order execution hides some of these latencies, but repeats scheduling work already done by the compiler and adds additional pipeline overhead. Other techniques, such as prefetching and multithreading, can hide some anticipable, long-latency misses, but not the shorter, more diffuse stalls due to difficult-to-anticipate, first or second-level misses. Our work proposes a microarchitectural technique, *two-pass pipelining*, whereby the program executes on two in-order back-end pipelines coupled by a queue. The "advance" pipeline often defers instructions dispatching with unready operands rather than stalling. The "backup" pipeline allows concurrent resolution of instructions deferred by the first pipeline allowing overlapping of useful "advanced" execution with miss resolution. An accompanying compiler technique and instruction marking further enhance the handling of miss latencies. Applying our technique to an Itanium 2-like design achieves a speedup of $1.38\times$ in *mcf*, the most memory-intensive SPECint2000 benchmark, and an average of $1.12\times$ across other selected benchmarks, yielding between 32 percent and 67 percent of an idealized out-of-order design's speedup at a much lower design cost and complexity.

Index Terms—Runahead execution, out-of-order execution, prefetching, cache-miss tolerance.

1 Introduction

Modern instruction set architectures offer the compiler several features for enhancing instruction level parallelism. Large register files grant the broad computation restructuring ability needed to overlap the execution latency of instructions. Explicit control speculation features allow the compiler to mitigate control dependences, further increasing static scheduling freedom. Predication enables the compiler to optimize program decision and to overlap independent control constructs while minimizing code growth. In the absence of unanticipated runtime delays such as cache miss-induced stalls, the compiler can effectively utilize execution resources, overlap execution latencies, and work around execution constraints [1]. For example, we have measured that, when runtime stall cycles are discounted, the Intel reference compiler can achieve an average throughput of 2.5 instructions per cycle (IPC) across SPECint2000 benchmarks for a 1.0GHz Itanium 2 processor.

- *R.D. Barnes is with George Mason University, 4400 University Drive, MS 1G5, Fairfax, VA 22030. E-mail: rbarnes1@gmu.edu.*
- *J.W. Sias is with Concordia Theological Seminary, Box 109, 6600 North Clinton Street, Ft. Wayne, IN 46825. E-mail: sias@crhc.uiuc.edu.*
- *E.M. Nystrom is with Universal Network Machines, 3255-3 Scott Blvd., Suuite 102, Santa Clara, CA 95054. E-mail: nystrom@crhc.uiuc.edu.*
- *S.J. Patel and W.W. Hwu are with the Department of Electrical and Computer Engineering, Coordinated Science Laboratory, MC 228, University of Illinois at Urbana-Champaign, 1308 West Main St., Urbana, IL 61801. E-mail: {sjp, hwu}@crhc.uiuc.edu.*
- *J. Navarro is with the Departament Arquitectura Computadors, Universitat Politecnica de Catalunya, Campus Nord, Modul D6, Jordi Girona 3, E-08034 Barcelona, Spain. E-mail: nacho@crhc.uiuc.edu.*

Manuscript received 19 Apr. 2004; revised 31 Jan. 2005; accepted 6 Apr. 2005; published online 22 Nov. 2005.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-0139-0404.

Interfering with the microarchitecture's task of exploiting the parallelism planned by the compiler, in the noted example reducing throughput to 1.3IPC, are various types of runtime stalls. This paper focuses on the bulk of those stall cycles—those that occur when a load misses in the cache and does not supply its result in time for its consumer to execute as scheduled. The execution time contribution of cache miss stall cycles is significant in the current generation of microprocessors and is expected to increase with the widening gap between processor and memory speeds [2]. Achieving high performance in any processor design requires that these events be mitigated effectively.

These problematic cache miss stall cycles can be costly to address either at compile or runtime. Compilers can attempt to schedule instructions according to their expected cache miss latencies, making the optimistic assumption that they can be predicted; such strategies, however, fail to capitalize on cache hits and can overstress critical resources such as machine registers. Out-of-order designs, on the other hand, rely on register renaming, dynamic scheduling, and large instruction windows to accommodate their latencies as they vary at runtime, sustaining useful computation during a cache miss. Although this general reordering effectively hides data cache miss delays, the mechanisms providing it replicate, at great expense, much work already done effectively by the compiler. Register renaming duplicates the effort of compile-time register allocation. Dynamic scheduling repeats the work of the compile-time scheduler. These mechanisms incur additional power consumption, add instruction pipeline latency, reduce predictability of performance, complicate EPIC feature implementation, and occupy substantial additional chip real estate.

Attempting to exploit the efficiencies of an in-order EPIC system while avoiding the penalty of cache miss stalls, this

paper details a new microarchitectural organization employing two in-order subpipelines bridged by a first-in-first-out buffer (queue). The "advance" subpipeline, referred to as the *A-pipe*, executes all instructions speculatively, stalling for unready input operands only on instructions marked as critical[1] by the compiler. Instructions dispatching without all of their input operands ready, rather than incurring stalls, are deferred, bypassing and writing specially marked nonresults to their consumers and destinations. Instructions whose operands are available execute normally. This propagation of nonresults in the A-pipe to identify instructions affected by deferral is similar to the propagation of NaT bits in the sentinel control speculation model [3], [4]. The "backup" subpipeline, the *B-pipe*, executes instructions deferred in the A-pipe and incorporates all results in a consistent order. This two-pipe structure allows cache miss latencies incurred in one pipe to overlap with independent execution and cache miss latencies in the other while preserving easy-to-implement in-order semantics in each. In keeping with the EPIC philosophy, the compiler's global view is used to classify instructions with respect to their "criticality" for the microarchitecture, improving its runtime effectiveness.

We argue that this *flea-flicker*[2] two-pass pipelining model [5] effectively hides the latency of many near cache accesses (such as hits in the L2 cache) and overlaps more longer-latency cache misses than an in-order design, providing substantial performance benefit while preserving the most important characteristics of EPIC design. Our simulations of SPECint2000 benchmarks characterize the prevalence and effects of the targeted latency events, demonstrate the effectiveness of the proposed model in achieving concurrent execution in the presence of these events, and illustrate the design issues involved in building two-pass systems.

2 Motivation and Case Study

A brief case study on a contemporary instruction-set architecture, Intel's Itanium, demonstrates the opportunities we intend to exploit. The Itanium ISA allows the compiler to express program parallelism using a wide-word encoding technique in which groups of instructions intended by the compiler to issue together in a single processor cycle are explicitly delimited. All instructions within such an "issue group" are essentially fused with respect to dependence-checking [4]. If an issue group contains an instruction whose operands are not ready, the entire group and all groups behind it are stalled. Thus, the program encoding generated by the compiler is not a general *specification* of available parallelism, but rather a particular *implementation* within that constraint. This design accommodates wide issue by reducing the complexity of the issue logic, but introduces the likelihood of "artificial"[3] dependences between instructions of unanticipated latency and instructions grouped with or subsequent to their consumers.

1. The notion of criticality, important to maintaining steady-state benefit in the two-pass model, will be described in Section 4.2.

2. In American football, the *flea-flicker* offense tries to catch the defense off gurad 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.

3. These dependences are artificial in the sense that they would not be observed in a dependence-graph-based execution of the program's instructions, as in an out-of-order microprocessor.

Not surprisingly, therefore, a large proportion of EPIC execution time is spent stalled waiting for data cache misses to return. When, for example, SPECint2000 is compiled with a commercial reference compiler (Intel ecc v.7.0) at a high level of optimization (-O3 -ipo -prof_use) and executed on a 1.0GHz Itanium 2 processor with 3MB of L3 cache, 38 percent of execution cycles are consumed by data memory access-related stalls. Furthermore, depending on the benchmark, between 10 percent and 95 percent of these stall cycles are incurred due to accesses satisfied in the second-level cache, despite its having a latency of only five cycles. As suggested previously, the compiler's carefully generated, highly parallel schedule is being disrupted by the injection of many, short, unanticipated memory latencies. The goal of the two-pass design is to absorb these events while allowing efficient exploitation of the compiler's generally good schedule.

Fig. 1 shows an example from one of the most significant loops in *mcf*, the SPECint2000 benchmark with the worst data cache behavior. The figure, in which each row constitutes one issue group and arrows indicate data dependences, shows one loop iteration plus one issue group from the next. In a typical EPIC machine, on the indicated cache miss stall caused by the consumption of r42 in group 1, all subsequent instructions (dotted box) are prevented from issuing until the load is resolved, although only those instructions enclosed in the solid box are truly dependent on the cache miss. (Since the last of these is a branch, the instructions subsequent to the branch are, strictly speaking, control dependent on the cache miss, but a prediction effectively breaks this control dependence.) An out-of-order processor would begin the processing of instructions such as the load in slot 3 of group 1 during the miss latency, potentially overlapping multiple cache misses. In an attempt to achieve such economy here, the compiler could explicitly schedule the code in such a way as to defer the stall (for example, by moving the consuming add after the load in slot 3 of group 1). However, in general, the compiler cannot anticipate statically which loads will miss and when. A limited degree of dynamic scheduling could easily overcome this problem, but too high a degree, in addition to unnecessarily complicating the design, would render an efficient implementation of EPIC features difficult to achieve. Even register renaming, a standard assumption of out-of-order design, substantially complicates the implementation of predicated execution, a fundamental feature of EPIC design [2].

3 The Two-Pass Pipeline Scheme

The proposed two-pass pipeline scheme is designed to allow the productive processing of independent instructions during the memory stall cycles left exposed in traditional in-order pipelines. Fig. 2a shows a snapshot of instructions executing on a stylized representation of a general in-order processor, such as Intel's Itanium or Sun's SPARC. In the figure, the youngest instructions are at the left; each column represents an issue group. A dependence checker determines if instructions have ready operands and are therefore ready to be dispatched to the execution engine. If any instruction is found not to be ready, its entire issue group is stalled. The darkened instructions in the dependence checker and incoming instruction queue are

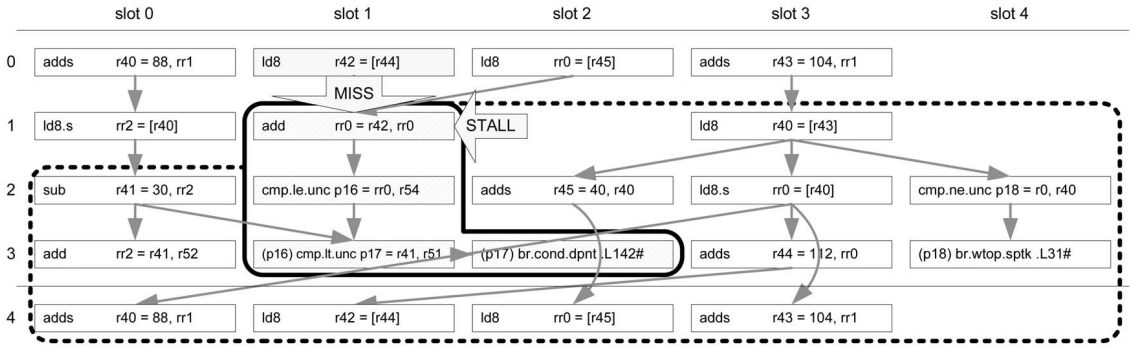


Fig. 1. Cache miss stall and artificial dependences.

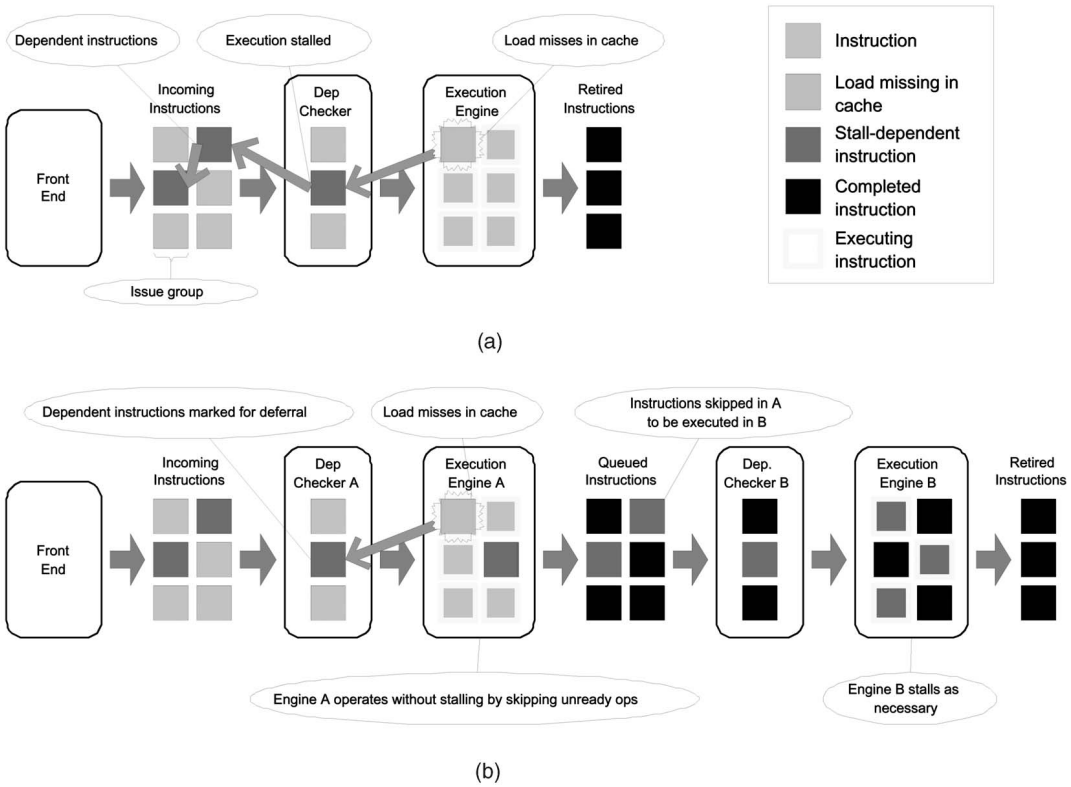


Fig. 2. Snapshot of execution: (a) original EPIC processor, (b) two-pass pipeline.

dependent on the stall, as indicated by arrows; the light-colored instructions, on the other hand, are dataflow-independent, but nonetheless stymied by the machine's issue group stall granularity.

Fig. 2b shows our proposed alternative. Here, when an input operand of instruction not marked "critical" by the compiler is unready at the dependence check, the processor, rather than stalling, marks the instruction and all *dependent* successors (as they arrive, through a propagation of "invalid" bits in the bypass network and register file) as deferred instructions. These are skipped by the first (A) execution engine. Subsequent *independent* instructions, however, continue to execute in engine A. Deferred instructions are queued up for processing in the second (B) engine, that always stalls when operands are not ready. Between the engines, instructions shown as blackened have begun execution; execution of the remaining instructions has been deferred in the A engine due to unavailable operands. These execute for the first time in the B engine, when their operands are ready. The B engine also incorporates into architectural state the results of instructions previously resolved in A. In the case of long or undetermined-latency instructions, such as loads, an instruction begun in the A engine may not be finished executing when its results are demanded in B; in this case, B must stall until the instruction completes. This situation is handled through the coupling mechanism to be described in the next section. This arrangement effectively separates the execution of the program into two concurrently executing streams: an "advance" stream, comprised of instructions whose operands are readily available at the first dispatch opportunity and those marked "critical," and a "backup" stream, encompassing the remainder. This section describes the proposed pipeline scheme in detail, focusing on the management of the two streams to maintain correctness as well as to maximize efficiency and concurrency.

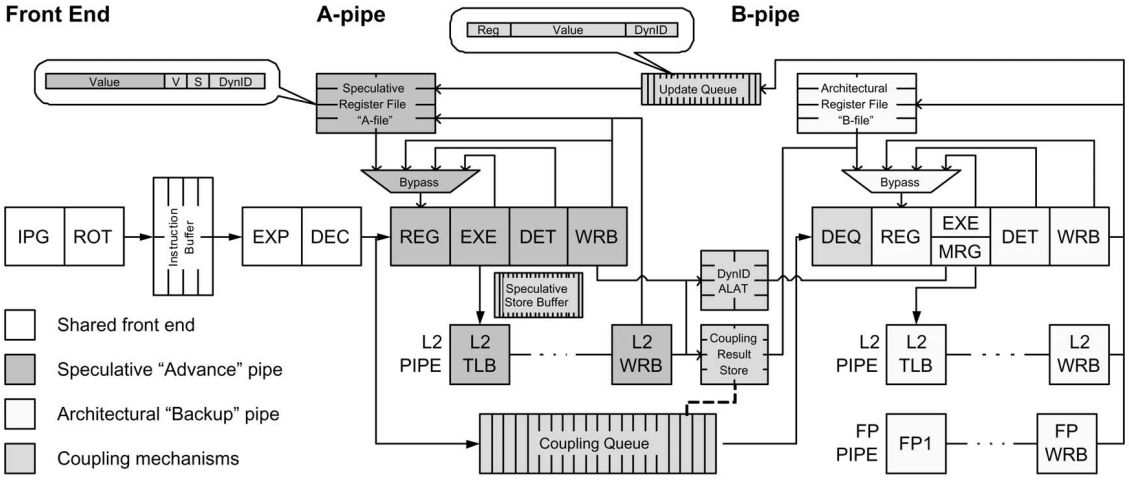


Fig. 3. Two-pass pipeline design.

3.1 Basic Mode of Operation

Fig. 3 shows a potential design of the proposed mechanism as applied to an architecture similar to Intel's Itanium.[4] The reader will note marked similarities between the front end and architectural pipeline and the pipeline of an Intel Itanium 2 microprocessor [4]. The remaining portions of the figure show the additions necessary to implement the proposed two-pass pipeline scheme. The front end portion (**I**nstruction **P**ointer **G**eneration, bundle **ROT**ation, bundle **EXP**ansion, and instruction **DEC**oding) prepares instructions for execution on the several functional units of the execution core.

The speculative pipeline in Fig. 3, referred to as the A-pipe, executes instructions on an issue group by issue group basis. Operands are read from the register file or bypass network in the **REG** ister read stage. An in-order machine stalls in this stage if the current issue group contains instructions with unready operands, often as the result of outstanding data cache misses. In a typical machine, these stalls consume a large fraction of execution cycles. In the proposed two-pass scheme, noncritical instructions found to be unready in the **REG** stage do not stall the processor; these instructions and their data-dependent successors are instead suppressed. Any subsequent, independent instructions are allowed to execute (**EXE**), detect exceptions and mispredictions (**DET**), and write their register results in the A-file (**WRB**).

The portion of Fig. 3 referred to as the B-pipe completes the execution of those instructions deferred in the A-pipe and integrates the execution results of both pipes into architectural updates. The A and B pipes are largely decoupled (e.g., there are no bypassing paths between them), contributing to the simplicity of this design. The B-pipe stage, **DEQ**, receives incoming instructions from the Coupling Queue (CQ), as shown in Fig. 3. The coupling queue receives decoded instructions as they proceed, in order, from the processor front end. When an instruction is entered into CQ, an entry is reserved in the Coupling Result Store (CRS) for each of its results (including, for stores, the value to be stored to memory). Instructions deferred in the A-pipe are marked as deferred in CQ and their corresponding CRS entries are marked as invalid. The B-pipe completes the execution of these deferred instructions.

When, on the other hand, instructions complete normally in the A-pipe, their results are written both to the A-file (if the target register has not been reused) and to the CRS. These "precomputed" values are incorporated in the merge (**MRG**) stage of the B-pipe, to be bypassed to other B-pipe instructions and written into the architectural B-file as appropriate. Scoreboarding on CRS entries handles "dangling dependences" due to instructions that begin in the the A-pipe but are not complete when they reach the B-pipe through the coupling queue. These instructions are allowed to dispatch in the B-pipe, but with scoreboarded destinations, to be unblocked when results arrive from the A-pipe. Through this mechanism, the need to reexecute (in the B-pipe) instructions successfully preexecuted (or even prestarted) in the A-pipe is obviated. This has two effects: First, it reduces pressure on critical resources such as the memory subsystem interface; second, since these preexecuted instructions are free of input dependences when they arrive in the B-pipe, an opportunity for height reduction optimizations is created. In an optimization described in [5], the B-pipe dispatch logic can regroup (without reordering) instructions as they are dequeued, allowing instructions from adjacent, independent instruction groups available at the end of the queue to issue together as machine resources allow.

As an example of the concurrency exposed by the two-pass technique, Fig. 4 shows four successive stages of execution of the code of Fig. 1 on the two-pass system. Instructions flow in vertical issue groups from the front end, on the left, into the A-pipe and coupling queue and then into the B-pipe. In Fig. 4a, a load issues in the A-pipe and misses in the first-level cache. In Fig. 4b, a dependent, noncritical instruction dispatches in the A-pipe. Since its operands are not ready, it is deferred and marked as such in the queue. A typical EPIC pipeline would have stalled issue rather than dispatch this instruction. In Fig. 4c, an additional dependent instruction is marked and a second cache miss occurs in the A-pipe. The concurrent processing of the two misses is enabled by the two-pass system. Finally, in Fig. 4d, two groups have retired from the A-pipe

4. Although the proposed pipeline scheme is described in the context of implementing the Intel Itanium Architecture, it could be applied similarly to other typically in-order architectures such as SPARC, ARM, and TI C6x.

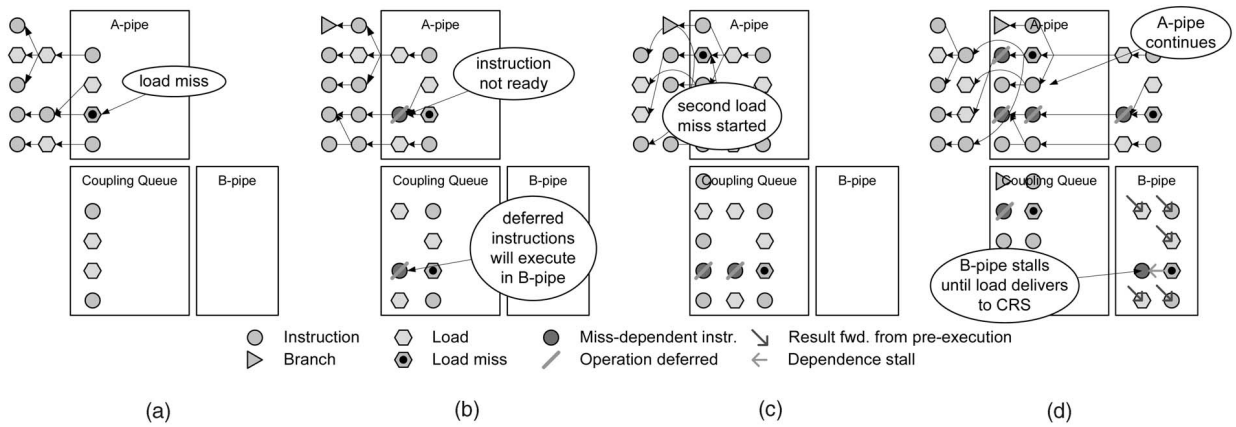


Fig. 4. Applying two-pass pipelining to the previous *mcf* example.

and reexecution has begun in the B-pipe. Many preexecuted instructions assume the values produced in the A-pipe, as propagated through the coupling result store. The original cache miss, on the other hand, is still being resolved and the inherited dependence causes a stall of the B-pipe. During this event, provided there is room in the coupling queue and result store, the A-pipe is still free to continue preexecuting independent instructions.

The coupling queue plays the important role of allowing instructions to wait for their operands without blocking the A-pipe. Based on empirical observations, the queue size was set to 64 instructions. The results were not particularly sensitive to reasonable variations in this parameter.

3.2 Critical Design Issues

Ensuring correctness and efficiency in the two-pass design requires the careful consideration of a number of issues. Chief among these is the fact that the B-pipe "trusts" the A-pipe in most situations to have executed instructions correctly; that is, the B-pipe does not confirm or reexecute instructions begun in the A-pipe, but merely incorporates their results. First, this entails that the A-pipe must accurately determine which instructions may be preexecuted and which must be deferred and must ensure that the A-file contains correct values for valid registers, even though write-after-write (WAW) stall conditions and other constraints typical to EPIC systems have been relaxed. Second, the proper (effective) ordering of loads and stores must be maintained, even though they are executed partially in the A-pipe and partially in the B-pipe. Finally, as the new pipe includes two stages at which the correct direction of a branch may be ascertained, the misprediction flush routine needs to be augmented. The following sections investigate these issues in detail.

3.3 Maintaining the A-File

The A-file, a speculative register file, operates in a manner somewhat unconventional to in-order EPIC designs, as delinquent instructions can write "invalid" results and as WAW dependences are not enforced by the A-pipe through the imposition of stalls (this is legitimate only because the B-file is the architectural one). Each register in the A-file is accompanied by a "valid" bit (V), set on the write of a computed result and cleared in the destination of an instruction whose result cannot be computed in the A-pipe, a "speculative" bit (S), set when an A-pipe instruction writes a result and reset when an update from the B-pipe arrives, and "DynID," a tag indicating the ID of the last dynamic instruction to write the register, sufficiently large to guarantee uniqueness within the machine at any given moment. The V bit supports the determination in the A-pipe of whether an instruction either has all operands available and, therefore, may execute normally or relies on an instruction deferred into the B-pipe and, therefore, must also be deferred to the B-pipe. The S bit marks those values written by the A-pipe but not yet committed by the B-pipe. All data so marked is speculative since, for example, a mispredicted branch might resolve in the B-pipe and invalidate subsequent instructions in the queue and the A-pipe, including ones that have already written to the A-file. This bit supports a partial update of the A-file as an optimization of the B-pipe flush routine, as discussed later, in Section 3.5. Finally, the dynamic ID tag (DynID) serves to allow the selective update of the A-file with results of retiring B-pipe instructions. An entry can be updated by a B-pipe retirement only if its outstanding invalidation was by the particular instruction retiring in the B-pipe on its deferral in the A-pipe.

3.4 Preserving a Correct and Efficient Memory Interface

Since the two-pass model can allow memory accesses to be performed out-of-order, the system must do some bookkeeping, beyond what is ordinarily required to implement consistency semantics, to preserve a consistent view of memory. To make a brief presentation of this issue, we consider representative pairs of accesses that end up having overlapping access addresses, but where program order is violated by the deferral of the first instruction to the B-pipe. Consider α to indicate an instruction that executes in the A-pipe and β one that executes in the B-pipe. Three dependence cases are of interest, as follows: Seemingly violated antidependences $\mathrm{ld}[addr]^{\beta} \overset{A}{\rightarrow} \mathrm{st}[addr]^{\alpha}$ and output dependences $\mathrm{st}[addr]^{\beta} \overset{O}{\rightarrow} \mathrm{st}[addr]^{\alpha}$ are resolved correctly due to the fact that loads and stores executing in the B-pipe do so with respect to architectural state and that stores executing in the A-pipe do not commit to this state, but, rather, write only to a speculative store buffer (an almost

ubiquitous microarchitectural element) for forwarding to A-pipe loads. When A-pipe stores reach the B-pipe, their results are committed, in order with other memory instructions, to architectural state.

Preserving a flow dependence $\mathrm{st}[addr]^{\beta} \xrightarrow{F} \mathrm{ld}[addr]^{\alpha}$ requires more effort. As indicated earlier, the general assumption is that instructions executed in the A-pipe either return correct values or are deferred. If, in the A-pipe, the store has unknown data but is to a known address, the memory subsystem can defer the load, causing it to execute correctly in the B-pipe, after the forwarding store. If, however, the store address is not known in the A-pipe, it cannot be determined in the A-pipe if the value loaded should have been forwarded or not. If a load executes in the A-pipe without observing a previous, conflicting store, the B-pipe must detect this situation and take measures to correct what (speculative) state has been corrupted. Fortunately, a device developed in support of explicit data speculation in EPIC machines, the Advanced Load Alias Table (ALAT) [6], [7] can be adapted to allow the B-pipe to detect when it is possible for such a violation to have occurred (since all stores are buffered, memory has not been corrupted). The Alpha 21264 [8], much in the same way, used a content-addressable memory to detect when a load dynamically reordered with a conflicting store.

Ordinarily, an advanced load writes an entry into an ALAT, a store deletes entries with overlapping addresses, and a check determines if the entry created by the advanced load remains. In two-pass pipelining, loads executed in the A-pipe create ALAT entries (indexed by dynamic ID rather than by destination register), stores executed in the B-pipe delete entries, and the merger of load results into the B-pipe checks the ALAT to ensure that a conflicting store has not intervened since the execution of the load in the A-pipe. If such a store has occurred, as indicated by a missing ALAT entry, the corrupted speculative state must be flushed (conservatively, all instructions subsequent to the load and all marked-speculative entries in the A-file), the A-file must be restored from the architectural copy, and execution must resume with the offending load. Since this can have a detrimental performance effect, the experimental results section presents material indicating the infrequency of these events. It should also be noted that this ALAT is distinct from any architectural ALAT for explicit data speculation and that, because of its cache-like nature, the ALAT carries the (small) possibility of false-positive conflict detections.

3.5 Managing Branch Resolution

Constructing a two-pass pipeline results in two stages where branch mispredictions can be detected: **A-DET** and **B-DET**, shown as the **DET** stages of the A-pipe and the B-pipe in Fig. 3. A conditional branch has subtly different execution semantics from other instructions in the A-pipe. When the direction of a branch cannot be computed in the execution stage of the A-pipe, the *mispredict detection* of the branch and not the effect of the branch itself is deferred to the B-pipe. This is implicit in the design of the pipelined machine since the branch prediction has already been incorporated into the instruction stream in the front end. When a branch misprediction is detected "early" in **A-DET**, the B-pipe continues to execute instructions preceding the branch until it "catches up" to the A-pipe by emptying the coupling queue. Any subsequent instructions present in the coupling queue, if any, must be invalidated, but, otherwise, fetch can be redirected and the A-pipe restarted as if the B-pipe were not involved. This can result in a reduction in observable branch misprediction penalties.

When mispredicted branches depend on deferred instructions for determination of either direction or target, however, the misprediction cannot be detected until the **B-DET** stage. In this case, the A-file may have been polluted with the results of wrong-path instructions beyond the misprediction. Consequently, all subsequent instructions in both the A-pipe and the B-pipe must be flushed, all corrupted state in the A-file must be repaired from the B-file, and fetch must be redirected. The "speculative" flags in the A-file reduce the number of registers requiring repair: Only the A-file entries marked as speculative need to be repaired from B-file data. As this procedure somewhat lengthens the branch misprediction recovery path for these instructions, performance may be degraded if too many misprediction resolutions are delayed to the B-pipe. Alternatively, one could employ a checkpoint repair scheme to enable faster branch prediction recovery at a higher register file implementation cost [9]. Various invalidation or A-file double-buffering strategies could also be applied.

3.6 Assessing the Implementation Cost

Justifying the use of both an advance and backup pipeline in the two-pass design requires a close evaluation of the complexity, power, and area required relative to that needed in an out-of-order design. Since neither two-pass pipelined nor out-of-order EPIC processors have yet been implemented, such comparisons are qualitative in nature and largely limited to the relative expense of analogous structures and rough estimations for structures unique to the two particular designs.

As stated in Section 3.1, the two-pass microarchitecture requires both an advance and an architectural register file. Additionally, each register in the advance file requires Valid and Speculative bits and a DynId. In contrast, typical out-of-order designs have a monolithic, physical register file with a size equal to the number of architected registers plus the planned number of in-flight instructions. Such a large capacity is critical to supporting effective renaming and a large instruction window and may be comparable in the number of registers to that used in a two-pass processor. Additionally, if an out-of-order processor utilizes a monolithic register file, such a register file is substantially larger and requires more power than two smaller register files [10], [11].

While the storage of the coupling queue (in terms of number of bits) might be larger than that of a reorder buffer because of the need to store complete instructions for processing in B-pipe, this queue is likely less complex than an out-of-order architecture's reorder buffer. While the A-pipe to B-pipe coupling structure is a simple first-in/first-out queue, reorder buffers are often implemented in power-hungry, associatively addressed storage [12]. While the coupling queue has only individual large read and write ports, the reorder buffer storage likely requires two read ports for every instruction issuing in a cycle and one write port for every instruction retiring.

Finally, both two-pass and out-of-order designs require hardware to ensure correctness in the presence of load and store reordering. The ALAT hardware is used in two-pass

pipelining and is similar in complexity to the content-addressable load and store buffers used by aggressive out-of-order implementations [8].

There are also hardware structures unique to each particular designs. For example, the two-pass model requires replication of the back-end pipelines and their functional units. There are two potentially significant sources of overhead from this replication. First, the doubling of the memory units implies additional data-cache read ports. The nonlinear relationship between cache size and porting makes this an expensive proposition. However, our design lends itself readily to partial replication. In particular, the A-pipe and B-pipe memory units could arbitrate for the same port resources. As shown in Section 5.3, the total number of memory accesses is very similar to a single pass approach because, despite two passes, no memory instruction accesses the cache twice. For this reason, there is only a negligible penalty for arbitrating load ports rather than replicating them. Second, some subpipelines, such as the floating-point subpipeline, can be fairly large. If necessary, arbitration can also be used, thereby allowing only partial replication or even complete sharing of any large functional units.

The update queue used by two-pass pipelining to feed retired values from the B-pipe back to the A-file is also a simple first-in/first-out queue. Furthermore, not only is this feedback path very localized and latency-tolerant, simplifying its design, but, as the results in Section 5.3 indicate, the majority of the benefit from a two-pass design can be achieved without this queue. Because this queue necessitated the DynID, eliminating the updates queue from the design could reduce significant complexity with a nominal degradation in performance. As register file access is often tied closely to cycle time, these DynIDs are the component of two-pass pipelining that is most likely to have cycle-time impact. When eliminated, the simple design of two-pass pipelining is unlikely to effect achievable processor frequency.

For the structures unique to two-pass pipelining, it is also important to consider the contribution of their cost with respect to the entire processor. First, the actual execution pipelines of a typical, contemporary microprocessor consume only a small fraction of the chip's transistor count and power consumption. For example, the Intel Itanium 2 integer pipeline and integer register file together consume an average of less than 5 percent of the total chip power and occupy less than 2 percent of the total chip area [13]. Even when the additional overhead of pipeline control is included, the impact of adding a second register file and an additional pipeline can be a reasonable trade-off for the cache miss tolerance that it provides. Implementing an EPIC ISA in an out-of-order design requires expensive support for features such as predication [14]; because flea-flicker's relatively independent pipelines internally maintain in-order semantics, it is a more natural model for implementation.

In contrast, the structures central to an out-of-order design can be very expensive. For example, in the POWER4, instruction schedulers and register renaming hardware alone account for more than 10 percent of total core power and the integer issue queue has the highest power density of any unit [15]. Though not modeled in the idealized out-of-order machine used for comparison in this work, the addition of these structures requires additional pipeline stages as well as new stall conditions (such as load data availability). Such overhead would likely be higher for an out-of-order EPIC design because of the relatively wide issue, the very large architectural register space, and the necessity to smoothly handle predicated instructions [14].

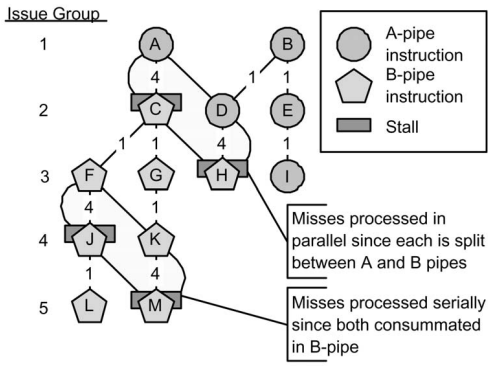


Fig. 5. Limitation on overlap of latency events.

4 Maximization of Miss Resilience

The two-pass model tolerates a miss by passing dependent consumers to the B-pipe for subsequent execution. Each such "tolerated" miss event potentially denies a dependent chain of operations the ability to tolerate misses since operations executing in B-pipe have no further pipeline to which to defer. This "single-second-chance" phenomenon limits the ability of two-pass to approach ideal out-of-order levels of performance and, if a deferred miss delays detection of a branch misprediction, may even degrade performance relative to traditional in-order execution. This section illustrates this problem and explains two mechanisms we apply to mitigate these effects: $B \rightarrow A$ update and compiler-based critical operation identification.

To illustrate the "single-second-chance" phenomenon, Fig. 5 shows a dependence graph including five issue groups, totaling 13 instructions. In the graph, each edge is marked with the dependence latency. Nonunit latencies on edges $A \rightarrow C$, $D \rightarrow H$, $F \rightarrow J$, and $K \rightarrow M$ indicate cache misses; each of these pairs, however, is scheduled at hit latency (one cycle). When A's result is not available for C's dispatch, C is deferred to B-pipe, as will be all its dataflow successors F, G, and so on. Because C deferred instead of stalling A-pipe, D and dependents can immediately begin execution in A-pipe, performing independent execution while C waits in B-pipe for its operand to be resolved. Miss latencies $A \rightarrow C$ and $D \rightarrow H$ are thus overlapped. Since instructions F and K execute in B-pipe, however, their unscheduled latencies to J and M, respectively, are exposed and serialized by B-pipe's in-order semantics; as deferred instructions, they cannot benefit from two-pass. Here, a greedy deferral strategy that never stalls the A-pipe would be outperformed by a strategy that sometimes chooses to expose a miss cost in A-pipe to preserve later benefit.

4.1 Keeping the A-File Up to Date with B-Pipe Results

One means of dealing with this limitation is to ensure timely updates of corrected state (and corresponding V-bits) from the architectural (B) register file to the A-file. This

allows dependent instructions to get the correct operands and execute in the A-pipe as long as their producers have executed in B-pipe by the time they go to dispatch in A-pipe. The DynID tag on each A-file register allows this update, as described in Section 3.3. In our initial design, every retirement in the B-pipe attempts to update the A-file. Since these updates may contend with retiring instructions in the A-pipe for A-file write ports, a buffer is provided, as shown in Fig. 3. Since, whenever a decision is made to defer the execution of an instruction to the B-pipe, it will not write the A-file, the bandwidth required is not expected to be much higher than in a traditional EPIC design. As the V-bits of these deferred instructions' destination registers in the A-file are cleared at dispatch time, the V-bits on these registers will defer all consumers until an update arrives from the B-pipe.

Without these running updates, the accumulation of invalid state in the A-file state might theoretically cause ever-increasing deferral to the B-pipe, although the A-file state is periodically completely updated in the restoration the follows detection of branch mispredictions in the B-file. Section 5.1 describes the performance consequences of the running $B \rightarrow A$ feedback path, showing that, while this feedback is useful for minimizing the number of instructions deferred to the B-pipe, the majority of the flea-flicker benefit can be achieved without resorting to the running update model.

4.2 Compiler-Based Critical Operation Identification

Updates from the B-pipe are only useful if they arrive before consumers execute in the A-pipe—assuming a large amount of slack in the schedule. When the dependence is tighter, as in the example of Fig. 5, however, this mechanism alone is insufficient. Once an instruction is deferred to the B-pipe, its data flow-dependent instructions will generally be deferred as well, with any unscheduled latency incurred being fully "materialized," or exposed in an in-order manner. Situations thus arise where it is better to stall the A-pipe to allow an instruction to receive its source operand rather than deferring the instruction to the B-pipe. This way, the latencies of its consumer instructions can be initiated in parallel from the A-pipe. (For example, stalling C in Fig. 5 allows parallel service of subsequent latencies $F \rightarrow J$ and $K \rightarrow M$.) While the hardware generally lacks the foreknowledge to determine when a stall should be materialized in the A-pipe to preserve later opportunities in the B-pipe, the compiler can easily be made to supply this crucial information.

Fig. 6 shows an example of a situation in which the compiler can distinguish between operations that should be deferrable and those "critical operations" that should always complete execution in A-pipe, in the interest of preserving future deferral opportunities. Fig. 6a shows three iterations of a stylized unrolled loop dependence graph in which arrows indicate dependences among the clouds of instructions. The arrow running through all iterations indicates a recurrence through the loop on which at least some parts of all subsequent iterations are dependent. Given the identification of operations participating in this recurrence, other loop operations can be classified as "prerecurrence," preceding the recurrence in the dependence graph, or "postrecurrence," dependent on recurrence operations. This classification is significant to the operation of two-pass, as shown in Fig. 6b. (It also is important in the comparison of two-pass to out-of-order execution, but this will be discussed later.) Here, the marked operation in the recurrence part of the first iteration is deferred to the B-pipe. This constrains all dataflow-dependent successors—all subsequent recurrence and post-recurrence operations—to be deferred as well. Any unscheduled latencies among these operations will be materialized in the B-pipe, degrading performance. The cost of these stalls may far outweigh the benefit of deferring the original recurrence-bound operation. Fig. 6c shows the result of preventing the deferral of operations participating in the recurrence. Here, a stall is materialized in A-pipe, slowing down the "advance" track, but the opportunity to absorb stalls among subsequent postrecurrence operations is preserved.

Preventing the deferral of all instructions that participate in recurrences can, however, detract from performance potential by delaying the execution of subsequent prerecurrence operations, as apparent in a comparison of Fig. 6b and Fig. 6c. Since these operations are not data-dependent on the recurrence, they are initiated without delay in the scheme of Fig. 6b. However, in Fig. 6c, the stall of the A-pipe will delay the A-pipe processing of the prerecurrence operations, thus reducing the ability of the two-pass pipeline to tolerate the latency of these instructions. There is thus a need to balance the desires to initiate subsequent operations early and to initiate a useful proportion of unscheduled-latency operations in the A-pipe.

These considerations lead to a straightforward compiler technique for marking operations for which, in the event of dispatch with unready operands, stalling in the A-pipe is a better strategy than deferring to the B-pipe. We assume the addition to each instruction of a single bit, a hint called an *audible*.[5] When this bit is set, marking an instruction "critical," and an operand is unready in the dependence check stage, the A-pipe stalls rather than deferring the instruction to the B-pipe. The compiler sets these bits in the following manner: For each procedure, all strongly connected components (recurrences) in the data dependence graph are identified. Each recurrence is evaluated to determine if its constituent operations should be marked with the "audible" bit, forbidding deferral. To do this, the compiler identifies, for each recurrence, the sets of pre and postrecurrence operations, according to the data dependence graph. The total execution weight (derived from a previous profiling run, part of our usual compilation path) of operations with latency-masking potential (loads and long-latency operations like multiplications and divisions) is computed for both sets. If the "masking potential" of the postrecurrence operations outweighs that of the prerecurrence operations, the recurrence operations are marked critical-not to be deferred. Otherwise, the benefit of starting prerecurrence operations early is judged to outweigh the cost of possibly deferring all postrecurrence operations and the recurrence is not specially marked. This heuristic was empirically demonstrated to deliver substantial gains on some benchmarks over an always-defer strategy, as indicated in Section 5.1. Table 2 shows the dynamic percentage of loads whose consumers are not marked critical, indicating that, for most benchmarks, a small

5. In American football, an *audible* is a verbal command by the quarterback to change an offensive play on short notice.

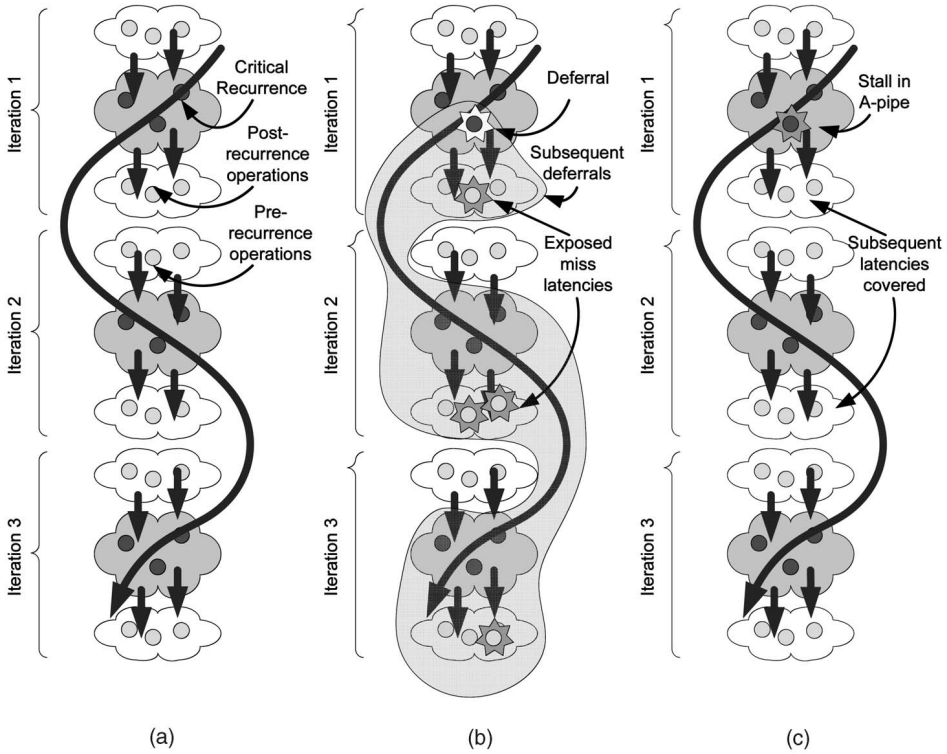


Fig. 6. Overlap of latency events preceding and succeeding critical recurrences. (a) Unrolled loop dependence graph showing a recurrence. (b) Poisoning effect of deferral on recurrence. (c) Benefit of preventing deferral.

minority of dynamic load latencies are prevented from being deferred using this technique.

4.3 Code Examples

To better discern how the two-pass pipelining microarchitecture exploits opportunities for cache latency tolerance, it is useful to further analyze examples of real program behavior. Three examples shown in Fig. 7, some of the most important loops in *mcf*, show varying potential for cache miss latency tolerance. In all three examples, execution time is dominated by frequent cache misses. The degree to which the flea-flicker mechanism can cover the cache miss latency in these examples is used to help explain the mechanism's benefits in Section 5.1. An additional example from *gap*, Fig. 8, illustrates the treatment of reductions as a special type of recurrence and shows how flea-flicker garners benefits besides cache miss latency. In these examples, loads are represented as hexagons, stores as squares, and branch instructions as triangles. Data dependence arcs reflecting cross-iteration dependences are dashed, rather than solid, lines.

Loop recurrences independent of cache misses. The first example in Fig. 7a shows the data dependence graph of a loop from function `primal_bea_mpp()` in *mcf*, in which each iteration depends simply on a stride computation, the recurrence highlighted in gray. In this example, loads 1 and 2 miss in the first-level cache on roughly half of their executions, while 3, 4, 5, and 6 virtually always miss. Since the stride computation, independent of all the cache misses, always executes in the A-pipe, loads 1, 2, 3, and 4, whose addresses it supplies, also execute in the A-pipe. As these loads incur misses, their dependents defer to the B-pipe; because the loop-driving recurrence is independent, such a deferral does not negatively affect future iterations of the loop. Loads 5 and 6, however, as dependents of loads 3 and 4 (which almost always miss), are usually deferred to the B-pipe. Their dependents suffer in-order stalls for lack of a third pipe to which to defer. Flea-flicker succeeds in overlapping subsequent execution with the latencies of loads 1, 2, 3, and 4 and, in so doing, hides most of the memory latency exposed in an in-order execution. Because it fails to hide misses in the dependence shadow of other misses, however, it falls short of an ideal out-of-order approach.

Loop recurrences containing cache misses. Fig. 7b, from `refresh_potential()`, has six miss-prone loads like the previous example, but, here, load 1 participates in the loop recurrence (again highlighted in gray). In this linked-list traversal, in which load 1 generates the pointer to the node to be processed in the next iteration, as long as the recurrence executes in the A-pipe, the next iteration's loads 1, 2, 3, and 4 can be initiated in the A-pipe, allowing misses on loads 2, 3, and 4 to be tolerated. As in the example from Fig. 7a, loads 5 and 6 are frequently deferred to the B-pipe because of the frequent misses in loads 2 and 3; thus, the B-pipe stall incurred by their dependents cannot be overlapped with misses from those instructions in future iterations. Since, in this example, load 1 participates in the recurrence, if its miss were to result in a deferral, all subsequent loads for the duration of the loop would execute in the B-pipe and would therefore have no latency tolerance ability. This recurrence is therefore marked by the compiler as critical to maintain a steady-state potential for benefit.

Loop with all cache misses entangled with the recurrence. Fig. 7c shows a loop from the function `price_out_impl()` containing five loads, all usually satisfied from the L3 cache. If an A-pipe initiated cache miss occurs in

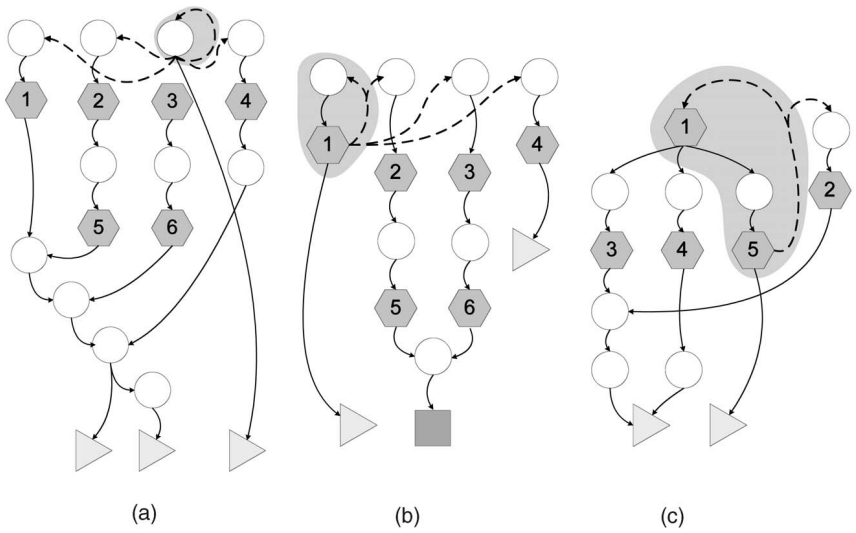


Fig. 7. Three example loops from *mcf*. (a) Strided recurrence. (b) Pointer dereference recurrence. (c) No misses disjoint from recurrence.

loads 2, 3, or 4 (not part of the recurrence), dependents can be deferred to the B-pipe and concurrent execution of the next iteration can commence in the A-pipe. In such a case, the two-pass mechanism would allow the overlap of the initial misses with cache misses in the next iteration. In reality, however, this loop does not yield a performance benefit on either the two-pass or out-of-order microarchitecture. As almost every dynamic load in this example is a cache miss, the latency tolerance mechanisms can do nothing to accelerate initiation of this loop beyond the sequential stalls caused by misses in load 1 and load 5. Additionally, the loads in this example access related memory locations. Different fields of only two different structures are accessed by these loads and, because of this relationship, load 1 and load 2 both access the one cache line while loads 3, 4, and 5 all access another. Therefore, no overlap of cache misses is possible from one iteration of the loop to the next. Time spent in data cache stall is thus not a universal indicator of potential for latency tolerance through even dynamic instruction scheduling; the dependence graph and relative data layout often limit benefit.

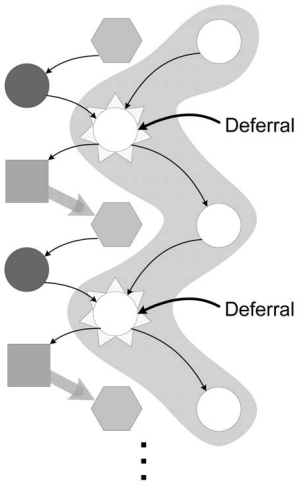


Fig. 8. Reduction in *gap*.

A reduction example. Fig. 8 shows yet another data dependence graph, this one from *gap*'s `ProdInt()`. In this example, loads almost always hit in cache. The execution time of this loop is dominated by the long-latency multiply instructions, shown as dark circles, and the loop recurrence is again indicated by the shaded portion. Shaded squares and wide arrows indicate stores and (spurious) memory flow dependences to subsequent (hexagonal) loads, respectively. These dependences reflect an inability of the compiler's pointer analysis to completely resolve the store and load addresses. In the compiled code, assuming explicit data speculation is not applied, the store-load dependences serialize the long latencies of the multiplies, resulting in a dramatic loss of in-order to out-of-order for this loop. Flea-flicker's compiler extension determines that this loop's recurrence is a reduction, one whose operations should be allowed to defer to capitalize on latency tolerance among the prerecurrence instructions. On the hardware, since the indicated operations defer to the B-pipe, rather than stalling, subsequent loads and multiplies can be initiated in the A-pipe while the B-pipe is waiting for the resolution of a given multiply instruction. This involves an implicit data speculation, as the subsequent load is initiated in the A-pipe before the store, dependent on the multiply and the recurrence and thus deferred to the B-pipe, completes. Here, the compiler's recurrence classification and the microarchitecture's deferral scheme work together to allow flea-flicker to achieve some of the benefit that an out-of-order model achieves by overlapping the long-latency multiplies.

5 Experimental Results

Baseline setup. In order to evaluate the two-pass design paradigm, we developed an in-order model, a two-pass model, and an idealized out-of-order model in a cycle-accurate, full-pipeline simulation environment. Table 1 shows the relevant machine parameters, which are derived from the Intel Itanium 2 design. This models an achievable near-term design; a futuristic design with longer cache latencies would further accentuate the demonstrated benefits. For the two-pass model, all nonmemory functional

TABLE 1
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
Two-pass Coupling Queue	64 entry
Two-pass ALAT	perfect (no capacity conflicts)
Out-of-Order Scheduling Window	64 entry
Out-of-Order Reorder Buffer	128 entry
Out-of-Order Scheduling and Renaming Stages	no additional stages
Out-of-Order Predicated Renaming	ideal

units are replicated in the advance pipeline and memory ports are arbitrated.

For our evaluation, eight benchmarks from SPECint2000 were compiled using the IMPACT compiler. Each program was optimized using control-flow profiling information (SPEC training inputs) and interprocedural points-to analysis. Optimizations applied included aggressive inlining, hyperblock formation, control speculation, modulo scheduling, and acyclic intrahyperblock instruction scheduling [16]. Results reflect rigorously sampled [17] complete runs of SPEC reference inputs.

Out-of-order model. The out-of-order model used for comparison with two-pass pipelining was constructed to give an idealized indication of the performance opportunities from dynamically ordering instructions. As mentioned in Section 3.6, a realistic out-of-order implementation would require additional front-end pipeline stages that would increase the branch misprediction penalty. Additionally, because of the multiple stages needed for instruction scheduling and register file read, modern instances of out-of-order execution require speculative wakeup and dispatch to allow back-to-back dispatch of a producing and consuming instruction. In our idealized model, no additional pipeline length is assumed and both scheduling and register file read are done in the **REG** stage, eliminating the need for speculative wakeup. Additionally, support for predication complicates renaming in an EPIC processor because multiple potential producers can exist for each consuming instruction. In our simulations, an ideal renamer was used, avoiding the performance cost of a realistic as described in like [14]. Finally, though, unconstrained reordering of loads and stores is performed in the scheduler of our out-of-order model. Since no prediction (oracle or realistic) is used to prevent loads from reordering with stores to unresolved addresses, there is the potential for costly data misspeculation flushes. As later examined in Section 5.3, in our presented results, these flushes only seriously impact one benchmark, *vpr*, almost doubling the number of front-end stalls.

5.1 Performance Experiments

Benchmark execution cycle counts are shown in Fig. 9 for baseline (**base**), two-pass pipelining (**2P**), and out-of-order (**OOO**) configurations, normalized to the number of cycles in the baseline machine. Results for *vpr*'s two inputs are presented separately as they differ somewhat in character. Within each bar, execution cycles are attributed to four categories: *unstalled execution*, in which instructions are issuing without delay, *front-end stalls*, including branch misprediction flushes and instruction cache misses, *other stalls*, those on multiplies, divides, floating-point arithmetic, and other non-unit-latency instructions, and *load stalls*, those on consumption of unready load results. For two-pass pipelining, these are measured in the B-pipe so that the architectural pipeline of the two-pass pipelined system is compared with that of the baseline. Cycles when out-of-order execution is not executing instructions 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).

For each benchmark, a significant number of memory stall cycles are eliminated by two-pass pipelining. This improvement results in a reduction in executed cycles for the **2P** system. For example, *mcf* shows a 34 percent reduction in memory stall cycles and a 27 percent reduction in overall cycles. Overall, two-pass pipelining reduces cache miss stalls by 35 percent relative to an in-order model. Not all of this reduction, however, results in performance increase, as the removal of a load-miss stall may occasionally expose another previously hidden stall. For example, in *vortex*, load stalls are reduced by 43 percent, while other stalls are increased by 62 percent, resulting in only a 5 percent net reduction in total cycles. Additionally, a small number of the eliminated load-miss stall cycles have been converted to fruitful execution cycles. The average reduction in total stall cycles (both load and nonload) is 24 percent, yielding our $1.12\times$ speedup.

The idealized out-of-order execution model reduces load stall cycles by an average of 48 percent and total stall cycles by 52 percent. The most substantial difference between **OOO** and **2P** load stall benefits occurs in *mcf*. As described in Section 4.3, the loops from *mcf* in Fig. 7a and Fig. 7b both contain two tiers of loads that are not part of the recurrence. While **2P** gets significant benefit from the ability to overlap cache misses in loads from the first tier, the miss penalties of the deferred, second-tier loads are serialized in the B-pipe. Out-of-order execution does not share the "single-second-chance" limitation and is free to overlap these misses as well. Where **2P** achieves a $2.1\times$ speedup on loop (a) and a $2.2\times$ speedup on loop (b), **OOO** achieves $3.4\times$ and $2.9\times$ speedups, respectively, in our experiments. This results in a $1.38\times$ benchmark speedup for **2P** in *mcf* compared to the potential $1.90\times$ total speedup shown by **OOO**.

The benefit of the compiler-based critical operation identification described in Section 4.2 is seen mostly clearly in *mcf*. Without the use of audible hint bits here, **2P** would only achieve a $1.13\times$ speedup over the baseline. *bzip* is the only other benchmark that achieves substantial benefit from the use of this technique, in which it increases the speedup

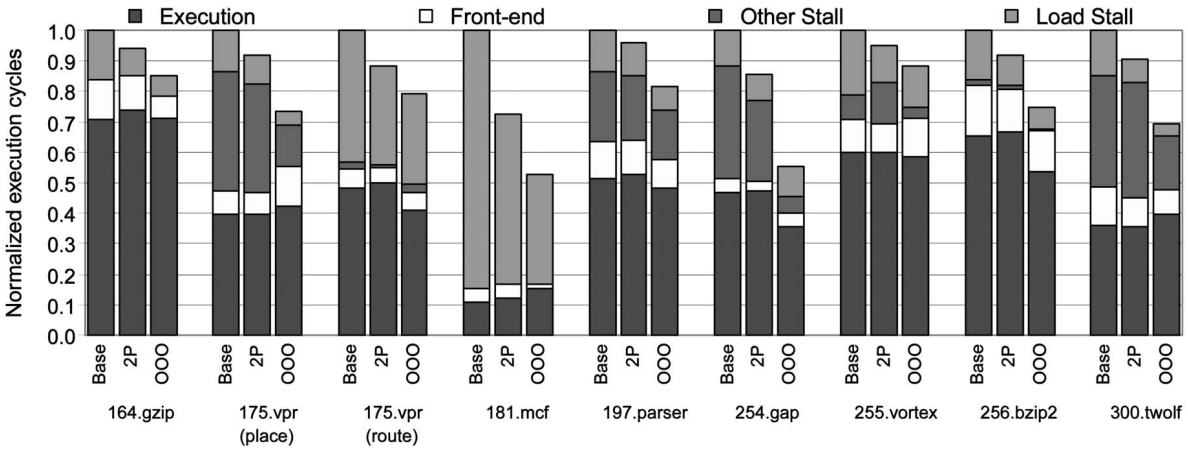


Fig. 9. Normalized execution cycles: baseline (base), two-pass (2P), and out-of-order (OOO).

from $1.06\times$ to $1.09\times$. For other benchmarks, these techniques make a difference of less than 1 percent in performance. In other benchmarks, either register-dependence recurrences are much less significant or they simply do not suffer frequently from chained cache misses.

Much of the remaining load stall time suffered by **OOO** (and by **2P**) represents the memory access latency that simply cannot be tolerated through execution of independent instructions. In the example from *mcf* in Fig. 7c, load misses occur on the critical path through the loop and there are no disjoint cache accesses that can begin early and be overlapped. In this case, **base** performs as well as **OOO** or **2P**.

The ability of two-pass pipelining to tolerate cache miss latency should not be unfairly overshadowed by the magnitude of performance improvement of **OOO** over **2P**. Where **OOO** benefit outstrips that of **2P**, this performance improvement often comes not from the memory tolerance targeted by **2P**, but from motion of instructions in the presence of compile-time scheduling barriers such as potential store to load memory dependences and control dependences [18].

As shown in the example from *gap* in Section 4.3, the reordering of potentially conflicting loads and stores can significantly reduce the schedule height and yield substantial performance improvement. While, in the example of Fig. 8, the previously serialized multiplies can be overlapped in the **2P** model, instructions are processed according to the original schedule and, thus, only one multiply can start every four cycles. Through dynamic scheduling, the **OOO** model can begin two multiplies each cycle (assuming the independent instructions have been fetched into its scheduling window). Thus, while this example highlights its ability to tolerate some nonload stall cycles, it also highlights its inability to move instructions earlier in the schedule. In another example, an almost identical loop in `SumInt()` from *gap*, a loop performing additions is serialized by false load-store dependences. In this case, there are no opportunities to defer instructions because addition is scheduled for its single cycle latency. Since the A-pipe executes instructions according to the original schedule, it can only defer instructions which are not ready; it cannot execute ready instructions any earlier than their compiler-specified placement, thus limiting the potential for **2P** benefit. In this example, while **OOO** reduces its execution cycles through overlapping several iterations of this loop, the loop executed in the **2P** model exactly as it would in the base.

Another benchmark exhibiting a nonmemory tolerance speedup on **OOO** that is much more significant than its reduction in load-stall cycles is the *place* input of *vpr*. The most significant loop occurs in `try_swap()` for this input. It spans 93 cycles and four separate hyperblocks, the last of which contains 30 cycles of floating-point computation that is not consumed by any further iteration of the loop. Trace expansion was performed in the compiler, but replication is limited by code expansion constraints. Because each hyperblock represents the compile-time "scheduling scope," the compiler was not able to hide the latency of this computation in the last hyperblock in the loop. **OOO** is able to overlap this computation with the next iteration of the loop. It is due to opportunities like this that **OOO** achieves a 65 percent reduction in nonmemory stalls.

This class of nonmemory benefits from out-of-order can be isolated through simulation of a perfect cache system. Since such a system has no opportunities for memory-latency tolerance, all speedups come from dynamically scheduling beyond the boundaries of the compiler's scheduling scope. The reduction in cycles in a perfect cache **OOO** system on the *place* input of *vpr* is equal to 67 percent of the reduction in the nonperfect cache system. For another benchmark with a large gap between **2P** and **OOO**, *bzip2*, the reduction in cycles in a perfect cache out-of-order system is equal to 74 percnet of the reduction in cycles on a nonperfect cache system. It is clear that **OOO** achieves significant benefits other than tolerance of unscheduled latency, while the flea-flicker technique is specifically only targeting these stalls. While the nonmemory benefits outweigh the cache-miss stalls in some benchmarks for current cache latencies, cache-related stalls are increasing as the latency to memory increases and are particularly problematic for EPIC designs. The other nonmemory opportunities are also already targeted by proposed mechanisms providing quasi-dynamic scheduling [19], [20], [21], static rescheduling of dynamic traces.

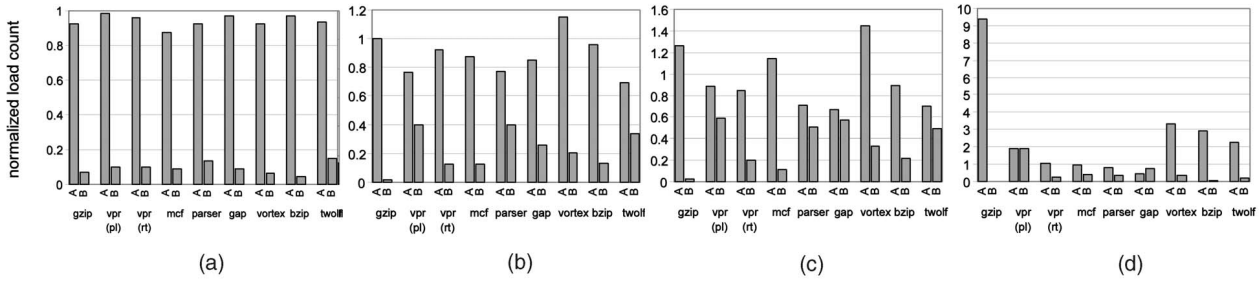


Fig. 10. Distribution of initiated accesses to A and B pipes, by satisfaction level. (a) L1 hits. (b) L2 hits. (c) L3 hits. (d) Main memory.

5.2 Critical Characteristics of Two-Pass Performance

Distribution of memory access initiations. Fig. 10 shows the distribution of the initiation of memory accesses to the A and B-pipes. Four charts are shown, one for each level of the cache hierarchy. To aid in interpreting this data, Table 2 shows, for each benchmark, the proportion of loads satisfied by each level of cache. For every benchmark, the majority of accesses are initiated in the A-pipe, indicating that it is largely successful in preexecuting loads, with a smaller portion of accesses being deferred to the B-pipe. Notable is the significant portion of the L3 cache misses in *mcf* started in the A-pipe. The benefit of overlapping the handling of these cache misses is clearly reflected in the performance improvement for *mcf*. *gap*, on the other hand, executes most of its substantial number of main memory accesses in the B-pipe and, thus, displays only a small performance improvement.

Because all execution in this pipeline is speculative (potentially down a wrong path of execution), there is a risk this speculative execution will result in an increase in the total number of memory accesses made. As can be seen in Fig. 10, this increase is, in general, a small percentage. The overall increase in number of loads is 5.4 percent. However, two-pass execution does result in a significantly increased number of accesses that go all the way to main memory. For example, *gzip* has more than a nine-fold increase in its small number of main memory accesses. These accesses that are not handled by any level of the cache are likely "wild loads" to invalid addresses, such as dereferencing a pointer off then end of an array or a linked-list traversal. In most cases, while the percentage increase in main memory accesses is significant, such accesses make up only a tiny fraction of all loads (0.1 percent for gap). Therefore, while these accesses are undesirable, they do not have a significant performance impact in our simulated results. Depending on the design of a realistic system, however such misses have the potential to cause performance-impacting TLB misses and faults.

Table 2 also presents the percentage of dynamic loads in each benchmark with a deferrable latency (i.e., the percentage of dynamic loads with no consumers marked critical). Marking too many consumers would keep the majority of load initiation in the A-pipe, but would eliminate opportunities for benefit. *mcf*, the benchmark with the largest improvement from the application of the "audible" hints, is also the benchmark with the lowest percentage of loads whose consumers can be deferred (68.5 percent). For the remaining benchmarks, more than 75 percent (and, in most cases, more than 90 percent) of the dynamic loads fall into this category. Therefore, the high rate of loads initiated in A-pipe is not simply the result of an over application of these hints.

Modes of benefit. We posit two modes of benefit from two-pass pipelining. First, as demonstrated in the examples from *mcf*, long latency memory instructions that would have otherwise been blocked by preceding stalled instructions can be started early in the two-pass system. This allows multiple long latency loads to be overlapped rather than processed sequentially. The second mode of benefit is that continuing execution beyond the consumer of a delinquent load allows the absorption of short cache misses. Since the code has been scheduled by a compiler assuming hit latencies, loads that miss in the first level of cache are often followed in quick succession by consuming instructions; in two-pass pipelining, these instructions can be deferred to the B-pipe, hiding the latency of these misses. Both of these techniques reduce the number of cycles in which the processor reports being stalled on load misses, as demonstrated in Fig. 9. When an application has poor cache locality, the benefit of overlapping long accesses dominates the benefit of hiding shorter ones (as in *mcf*). For other benchmarks, like *gzip*, there are relatively few long latency misses. The performance gain seen in **2P** for *gzip* is likely due to the second source, the absorption of latencies from short but ubiquitous misses.

Negative performance effects. Other than the potential increase in memory traffic, two-pass execution has the potential to degrade performance in two other situations. First, it extends the effective pipeline length for any misprediction detected in the B-pipe, increasing misprediction recovery cost. In our simulations, an average of 68 percent of branch mispredictions are discovered and repaired in the A-pipe. The effects of these mispredictions are less severe than in the single-pipe design as the B-pipe may continue to process instructions during the redirection

TABLE 2
Distribution of Cache Access to Satisfying Level and Percentage of Load Latencies Deferrable

Benchmark	Memory access satisfaction levels L1	L2	L3	MM	Dynamic proportion of non-critical loads
gzip	81.8%	18.1%	0.0%	0.1%	74.7%
vpr (place)	78.2%	15.4%	6.4%	0.0%	84.6%
vpr (route)	52.8%	40.7%	5.1%	1.4%	84.6%
mcf	57.1%	19.6%	17.8%	1.5%	68.5%
parser	91.1%	7.1%	1.7%	0.1%	90.0%
gap	97.7%	1.8%	0.2&	0.3%	97.5%
vortex	94.0%	4.5%	1.2%	0.3%	99.9%
bzip2	95.5%	2.5%	1.8%	0.2%	91.75%
twolf	82.2%	9.7%	8.1%	0.0%	96.77%

of the A-pipe as long as the coupling queue has instructions remaining. Second, store-conflict flushes are incurred whenever a store initiated in the B-pipe conflicts with a programmatically subsequent load that was already initiated in the A-pipe, as discussed in Section 3.4. Initiating loads in the A-pipe (even in the presence of deferred, ambiguous stores) is advisable, as 98.7 percent of all load accesses initiated in the A-pipe while a deferred store is in the queue are free of store conflicts. Overall, only 0.2 percent of all stores are deferred to the B-pipe and eventually cause a conflict flush.

Taking into account misprediction and store flushes, the results of Fig. 9 show by the reasonable size of the front-end stall segment that neither substantially erodes the performance gained from two-pass pipelining. In fact, by executing branches that were positionally blocked by cache-miss stalls, **2P** often sees a reduction in front-end stall time. **OOO** in general sees an even greater reduction in front-end stalls with the notable exception of *vpr*. The more aggressive reordering of instructions in **OOO** resulted in 10 times as many load/store conflict flushes in the **OOO** compared to the **2P** model. This has resulted in close to a doubling of the front-end stall cycles over the baseline for *vpr*. While the idealized **OOO** shows an average decrease in front-end stalls, this would be offset in a realistic implementation by its additional pipeline stages.

Finally, continued successful preexecution might require that committed results in the B-pipe be fed back in a timely manner into the A-pipe to prevent the deferral of ever-greater numbers of instructions. As this interpipe communication may be difficult to implement in a single cycle, we evaluated the effect of latency on this update path and found that, in general, not only is the update queue tolerant to latency, the total omission of update had a small impact on performance. The benchmark most impacted by the lack of A-file update was *mcf*, which saw less than a 4 percent increase in cycles with the removal of the update queue.

Comparison to previous two-pass experiments. There are two important distinctions that should be made between the experiments presented here and those previously presented for the two-pipelining technique [5]. First, in this work, we have utilized the reference inputs for each of the benchmarks rather than a reduced-length input. In general, a greater number of cache misses is seen with the reference inputs and a greater number of misses are dependent upon other misses. Because the reference inputs exhibit a greater number of load-stall cycles, a larger speedup is seen for the **2P** model for most benchmarks. In the experiments presented here, the compiler technique described in Section 4.2 is introduced. Because of the increased frequency of miss-dependent misses, without that technique *mcf* would exhibit a dramatic increase in deferred loads and a sharp reduction in speedup. The necessity of these updates is likely lessened by the prevention of critical operations from being deferred. Second, an effort has been made to improve both the realism of the machine model and the quality of the compiled code used in our experiments. A machine model much closer to the contemporary Itanium 2 processor, in terms of functional units and cache sizes and latencies, has been implemented. To improve the baseline performance of the code, thereby minimizing nonmemory tolerance opportunities, the code used for these experiments has been optimized much more effectively. The most important of the additional optimizations performed is modulo scheduling, which eliminates most opportunities for out-or-order execution to find scheduling benefit around the back-edge of tight loops.

With these noted differences, the speedups shown in this work are roughly comparable to those in [5] and the conclusions drawn about the performance opportunity presented by the two-pass pipeline technique are the same. One final distinction with the previous results is that the technique described in [5] in which new issue groups are formed in the B-pipe (exploiting computation already performed in the A-pipe) provided a significant portion of the benefits presented there. In our current experiments, the additional benefit provided by this technique has been reduced and it is not utilized for the presented results. A potential explanation is that the technique achieves some of its benefits at compiler-block boundaries and because of the increased quality of the experimental code, base opportunities for these benefits have been reduced.

6 Related Work

This paper is not alone in proposing a mechanism for improving the tolerance of variable-latency instructions. Its uniqueness is that it targets relatively short cache miss latencies, which cause a pronounced but distributed performance problem for EPIC machines, while maintaining the inherent simplicity of an EPIC core design.

Dundas and Mudge [22] and Mutlu et al. [23] both propose run-ahead schemes relying on checkpointing and repair. Mutlu et al. present a run-ahead implementation specifically targeting long-latency misses in out-of-order machines. The technique attempts to accommodate such misses efficiently without overstressing out-of-order instruction scheduling resources. In a single-issue, short-pipeline, in-order machine, Dundas and Mudge examine run-ahead execution in a special mode during any L1 cache miss. In Dundas and Mudge's model, run-ahead begins when a cache miss occurs, not, as in this work, when the consuming instruction executes. In cases where the consumers of a load are scheduled farther away than the load's hit latency, Dundas and Mudge's mechanism could enter run-ahead unnecessarily.

Both these approaches discard results of run-ahead execution (aside from memory accesses initiated) when the run-ahead mode is terminated. Our mechanism, on the other hand, not only allows the correct portion of the run-ahead execution to be retained, but also allows for the simultaneous execution of both run-ahead and standard instruction streams and provides for the feedback of architectural thread computation to the run-ahead thread. In a manner similar to our approach, both run-ahead designs utilize a second register file during run-ahead mode. Since run-ahead work is not preserved, register state is repaired at the end of each runahead effort. Additionally, to avoid refetching once a run-ahead-activating load has completed, these previous run-ahead techniques also would require additional instruction queues. Thus, the added cost of our approach relative to these techniques is limited to the replication of the execution pipeline and the addition of memory dependence detection hardware.

Slipstream processors [24] and master/slave speculative parallelization [25] attempt to exploit additional instruction-

level parallelism by selecting program threads for preexecution. Slipstream does this dynamically by squashing predictably useless instructions out of the "advance" stream. Master/slave uses multiple "slave" checkers to verify, in parallel, sections of the "master's" execution of a "distilled" version of the program. These approaches have in common with ours the strategy of achieving better parallelism through partitioned program execution. As in our approach, the leading thread performs persistent program execution; these systems, however, use a much coarser mechanism for partitioning program streams than two-pass pipelining's fine-grained, cycle-by-cycle mechanism. Unlike these thread approaches that attempt to execute all useful work in the leading thread, our technique specifically defers useful computation to avoid stalling the leading, in-order thread on the consumers of load misses.

Other related approaches share some of our goals, but differ significantly in approach. Simultaneous subordinate microthreading [26] adds additional microthreads, the sole purpose of which is to help the microarchitecture execute the main thread more efficiently. Like the run-ahead architectures, these threads can initiate memory accesses early with the goal of reducing the cache stalls of the main thread. Decoupled architectures [27] also allow the latencies of loads to be overlapped by issuing all loads separately from their consumers and then delivering their results through a architecturally visible data queue.

Collins et al. [28], [29] proposed software-based speculative precomputation and prefetching targeted to particular delinquent loads. Annavaram et al. [30] proposed a dynamic mechanism to generate prefetching microthreads for preexecution. These techniques, because they require code generation or dynamic slice extraction for specific delinquent loads, address a different problem than the diffuse serialization of occasional misses targeted here.

7 Conclusion and Future Work

We have described the "flea-flicker" microarchitecture, a novel two-pipeline organization designed to carry out useful work during unscheduled latencies while retaining the basic simplicity of an in-order pipeline model. With detailed simulations, we show how an implementation of this technique achieves roughly half the total stall cycle reduction of an idealized out-of-order design in an Itanium 2-like EPIC machine. This result, reflecting a $1.12\times$ average speedup over in-order and $1.38\times$ on *mcf*, is significant as real out-of-order implementations for such an EPIC ISA would be heavily penalized due to features such as predication and an already-large architectural register space, as well as pipeline and power constraints, shrinking the apparent gap between flea-flicker and OOO. We expect our benefits to scale favorably as substantial storage recedes from the processor due to scaling disparity.

Analysis and examples of SPECint2000 benchmarks illumine the effects of flea-flicker and OOO in different dependence contexts, showing how each's potential is impacted by miss interdependence. This analysis led to compiler-placed annotations that tripled our gain on *mcf*, the most miss-laden component of SPECint2000, by helping the microarchitecture maintain a steady-state opportunity for benefit in important loops. Our results further show much of the performance gap between our technique and OOO to derive from non-miss-related instruction motion benefits. This suggests that a combination of flea-flicker with one of the several existing quasi-dynamic scheduling techniques could provide a very efficient and relatively comprehensive alternative to an out-of-order model. Finally, opportunities exist to reduce replication cost and to mitigate the "single-second-chance" phenomenon by applying the flea-flicker paradigm to a single-shared pipeline providing an arbitrary number of virtual "passes."

Acknowledgments

This work was supported by the MARCO/DARPA Gigascale Systems Research Center and the Center for Circuits, Systems, and Solutions and the US National Science Foundation Information Technology Research program under contract number 0086096. Nacho Navarro has been partially supported by the Ministry of Science and Technology of Spain and by the European Union (FEDER) under contract TIC 2001-0995-C02-01 and the HiPEAC European Network of Excellence. The authors thank John Crawford, John Shen, Ed Grochowski, Joel Emer, Chris Newburn, and Matthew Merten at Intel Corporation and Jim McCormick at Hewlett-Packard for their generous and insightful feedback. They also thank Shane Ryoo and Christopher Rodrigues for their debugging and simulation efforts and the anonymous reviewers for their helpful comments.

References

[1] D.I. August, D.A. Connors, S.A. Mahlke, J.W. Sias, K.M. Crozier, B.-C. Cheng, P.R. Eaton, Q.B. Olaniran, and W.W. Hwu, "Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture," *Proc. 25th Ann. Int'l Symp. Computer Architecture,* pp. 227-237, July 1998.

[2] P.H. Wang, H. Wang, J.D. Collins, E. Grochowski, R.M. Kling, and J.P. Shen, "Memory Latency-Tolerance Approaches for Itanium Processors: Out-of-Order Execution vs. Speculative Precomputation," *Proc. Eighth Int'l Symp. High-Performance Computer Architecture,* pp. 167-176, Feb. 2002.

[3] S.A. Mahlke, W.Y. Chen, R.A. Bringmann, R.E. Hank, W.W. Hwu, B.R. Rau, and M.S. Schlansker, "Sentinel Scheduling: A Model for Compiler-Controlled Speculative Execution," *ACM Trans. Computer Systems (TOCS),* vol. 11, no. 4, pp. 376-408, 1993.

[4] Intel Corp., *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization,* Apr. 2003.

[5] R.D. Barnes, E.M. Nystrom, J.W. Sias, S.J. Patel, N. Navarro, and W.W. Hwu, "Beating In-Order Stalls with 'Flea-Flicker' Two-Pass Pipelining," *Proc. 36th Ann. Int'l Symp. Microarchitecture,* pp. 387-398, Nov. 2003.

[6] D.M. Gallagher, W.Y. Chen, S.A. Mahlke, J.C. Gyllenhaal, and W.W. Hwu, "Dynamic Memory Disambiguation Using the Memory Conflict Buffer," *Proc. Sixth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 183-193, Oct. 1994.

[7] R. Zahir, J. Ross, D. Morris, and D. Hess, "OS and Compiler Considerations in the Design of the IA-64 Architecture," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 213-222, Oct. 2000.

[8] R.E. Kessler, "The Alpha 21264 Microprocessor," *IEEE Micro,* vol. 19, Mar./Apr. 1999.

[9] W.W. Hwu and Y.N. Patt, "Checkpoint Repair for Out-of-Order Execution Machines," *Proc. 14th Ann. Int'l Symp. Computer Architecture,* pp. 18-26, July 1987.

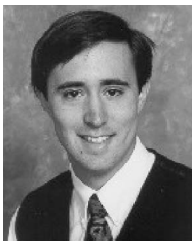
[10] V. Zyuban and P. Kogge, "The Energy Complexity of Register Files," *Proc. 1998 Int'l Symp. Low Power Electronics and Design,* pp. 305-310, Aug. 1998.

[11] V. Agarwal, M.S. Hrishikesh, S.W. Keckler, and D. Burger, "Clock Rate versus IPC: The End of the Road for Conventional Microprocessors," *Proc. 27th Ann. Int'l Symp. Computer Architecture,* pp. 248-259, July 2000.

[12] D. Ponomarev, G. Kucuk, and K. Ghose, "Reducing Power Requirements of Instruction Scheduling through Dynamic Allocation of Multiple Datapath Resources," *Proc. 34th Ann. Int'l Symp. Microarchitecture,* pp. 90-101, Nov. 2001.
[13] E.S. Fetzer, M. Gibson, A. Klein, N. Calick, C. Zhu, E. Busta, and B. Mohammad, "A Fully Bypassed Six-Issue Integer Datapath and Register File on the Itanium-2 Microprocessor," *IEEE J. Solid-State Circuits,* vol. 37, Nov. 2002.
[14] P.H. Wang, H. Wang, R.M. Kling, K. Ramakrishnan, and J.P. Shen, "Register Renaming and Scheduling for Dynamic Execution of Predicated Code," *Proc. Seventh Int'l Symp. High-Performance Computer Architecture,* pp. 15-25, Jan. 2001.
[15] P. Bose, D. Brooks, A. Buyuktosunoglu, P. Cook, K. Das, P. Emma, M. Gschwind, H. Jacobson, T. Karkhanis, P. Kudva, S. Schuster, J.E. Smith, V. Srinivasan, and V. Zyuban, "Early-Stage Definition of LPX: A Low-Power Issue-Execute Processor," *Proc. Second Int'l Workshop Power-Aware Computer Systems,* pp. 1-17, 2003.
[16] J.W. Sias, S.-Z. Ueng, G.A. Kent, I.M. Steiner, E.M. Nystrom, and W.W. Hwu, "Field Testing IMPACT EPIC Research Results in Itanium 2," *Proc. 31st Ann. Int'l Symp. Computer Architecture,* June 2004.
[17] R.E. Wunderlich, T.F. Wenisch, B. Falsafi, and J.C. Hoe, "SMARTS: Accelerating Microarchitectural Simulation via Rigorous Statistical Sampling," *Proc. 30th Ann. Int'l Symp. Computer Architecture,* pp. 84-95, June 2003.
[18] R.D. Barnes, J.W. Sias, E.M. Nystrom, and W.W. Hwu, "EPIC's Future: Exploring the Space between In- and Out-of-Order," *Proc. Third Workshop Explicitly Parallel Instruction Computing Architectures and Compiler Technology,* Mar. 2004, http://www.cgo.org/html/workshops/epic3_program.htm.
[19] M.C. Merten, A.R. Trick, R.D. Barnes, E.M. Nystrom, C.N. George, J.C. Gyllenhaal, and W.W. Hwu, "An Architectural Framework for Run-Time Optimization," *IEEE Trans. Computers,* vol. 50, no. 6, pp. 567-589, June 2001.
[20] F. Spadini, B. Fahs, S.J. Patel, and S.S. Lumetta, "Improving Quasi-Dynamic Schedules through Region Slip," *Proc. First Int'l Symp. Code Generation and Optimization,* pp. 149-158, Apr. 2003.
[21] H. Chen, W.-C. Hsu, and D.-Y. Chen, "Dynamic Trace Selection Using Performance Monitoring Hardware Sampling," *Proc. First Code Generation and Optimization,* pp. 79-90, 2003.
[22] J. Dundas and T. Mudge, "Improving Data Cache Performance by Pre-Executing Instructions under a Cache Miss," *Proc. 11th Ann. Int'l Conf. Supercomputing,* pp. 66-75, June 1997.
[23] O. Mutlu, J. Stark, C. Wilkerson, and Y.N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-Order Processors," *Proc. Ninth Int'l Symp. High-Performance Computer Architecture,* pp. 129-140, Feb. 2003.
[24] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "A Study of Slipstream Processors," *Proc. 33rd Ann. Int'l Symp. Microarchitecture,* pp. 269-280, Nov. 2000.
[25] C. Zilles and G. Sohi, "Master/Slave Speculative Parallelization," *Proc. 35th Ann. Int'l Symp. Microarchitecture,* pp. 85-96, Nov. 2002.
[26] R.S. Chappell, J. Stark, S.P. Kim, S.K. Reinhardt, and Y.N. Patt, "Simultaneous Subordinate Microthreading (SSMT)," *Proc. 26th Ann. Int'l Symp. Computer Architecture,* pp. 186-195, July 1999.
[27] J.R. Goodman, J. Hsieh, K. Liou, A.R. Pleszkun, P. Schechter, and H.C. Young, "PIPE: A VLSI Decoupled Architecture," *Proc. 12th Ann. Int'l Symp. Computer Architecture,* pp. 20-27, July 1985.
[28] J.D. Collins, H. Wang, D.M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J.P. Shen, "Speculative Precomputation: Long-Range Prefetching of Delinquent Loads," *Proc. 28th Ann. Int'l Symp. Computer Architecture,* pp. 14-25, July 2001.
[29] J.D. Collins, D.M. Tullsen, H. Wang, and J.P. Shen, "Dynamic Speculative Precomputation," *Proc. 34th Ann. Int'l Symp. Microarchitecture,* pp. 306-317, Nov. 2001.
[30] M. Annavaram, J.M. Patel, and E.S. Davidson, "Data Prefetching by Dependence Graph Precomputation," *Proc. 28th Ann. Int'l Symp. Computer Architecture,* pp. 52-61, July 2001.

Ronald D. Barnes received the BS degree in electrical engineering from the University of Oklahoma and the MS and PhD degrees in electrical engineering from the University of Illinois at Urbana-Champaign. His research interests include complexity-effective microarchitectures, program-phase detection, and binary and runtime optimization. He is a member of both the IEEE and the ACM. He is currently an assistant professor at George Mason University in Fairfax, Virginia.

John W. Sias received the BS degree in computer engineering and the MS degree in electrical engineering from the University of Illinois at Urbana-Champaign in 1997 and 1999, respectively. His research focuses on compiler technology for enhancing instruction-level parallelism in EPIC architectures. He expects to complete the PhD degree in electrical engineering in 2005. He is a student member of the IEEE.



Erik M. Nystrom received the BS degree in computer engineering from North Carolina State University and the MS and PhD degrees in electrical engineering from the University of Illinois at Urbana-Champaign. He is currently involved in compiler development for Universal Network Machines in Santa Clara, California. His research interests include pointer analysis algorithms, software pipelining, and dynamic optimization. He is a student member of the IEEE.

Sanjay J. Patel received the PhD degree in computer science and engineering from the University of Michigan, Ann Arbor. He is an assistant professor of electrical and computer engineering and Willett Faculty Scholar at the University of Illinois at Urbana-Champaign. His research interests include processor microarchitecture, computer architecture, and high-performance and reliable computer systems. He is currently serving as chief architect at AGEIA Technologies. He is a member of the IEEE.

Jose (Nacho) Navarro received the PhD degree in computer science from the Universitat Politecnica de Catalunya. He is a visiting research professor at the University of Illinois at Urbana-Champaign and associate professor at UPC, Barcelona, Spain. His research interests include operating systems for high performance computing and optimization of runtime environments for embedded architectures. He is a member of the ACM, IEEE, and USENIX associations.

Wen-mei W. Hwu received the PhD degree in computer science from the University of California, Berkeley. He is the Sanders-AMD Endowed Chair Professor of the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign. His research interest is in the area of architecture, implementation, and software for high-performance computer systems. He is the director of the IMPACT lab, which has delivered new compiler and computer architecture technologies to the computer industry since 1987. He is a fellow of the IEEE and the ACM.