Beating in-order stalls with "flea-flicker"*two-pass pipelining

Ronald D. Barnes Erik M. Nystrom John W. Sias
Sanjay J. Patel Nacho Navarro Wen-mei W. Hwu
Center for Reliable and High-Performance Computing
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
{rdbarnes, nystrom, sias, sjp, nacho, hwu}@crhc.uiuc.edu

Abstract

Accommodating the uncertain latency of load instructions is one of the most vexing problems in in-order microarchitecture design and compiler development. Compilers can generate schedules with a high degree of instruction-level parallelism but cannot effectively accommodate unanticipated latencies; incorporating traditional out-of-order execution into the microarchitecture hides some of this latency but redundantly performs work done by the compiler and adds additional pipeline stages. Although effective techniques, such as prefetching and threading, have been proposed to deal with anticipable, long-latency misses, the shorter, more diffuse stalls due to difficult-to-anticipate, first- or second-level misses are less easily hidden on in-order architectures. This paper addresses this problem by proposing a microarchitectural technique, referred to as two-pass pipelining*, wherein the program executes on two in-order back-end pipelines coupled by a queue. The "advance" pipeline executes instructions greedily, without stalling on unanticipated latency dependences (executing independent instructions while otherwise blocking instructions are deferred). The "backup" pipeline allows concurrent resolution of instructions that were deferred in the other pipeline, resulting in the absorption of shorter misses and the overlap of longer ones. This paper argues that this design is both achievable and a good use of transistor resources and shows results indicating that it can deliver significant speedups for in-order processor designs.*

1 Introduction

Modern instruction set architectures offer the compiler several features supporting the enhancement of instruction-level parallelism and the generation of aggressive schedules for wide issue processors. Large register files grant the compiler 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 run-time 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 run-time 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.

Run-time stall cycles of various types prolong the execution of the compiler-generated schedule, in the noted example reducing throughput to 1.3IPC. This paper focuses on the majority of those stall cycles—those that arise due to a load instruction missing in the data cache, when a load's result does not arrive in time for consumption by its consumer instruction, triggering an interlock. Cache miss stall cycles are significant in the current generation of microprocessors and are expected to increase as the gap between processor and memory speeds continues to grow [2]. Achieving high performance in any processor design requires that they be mitigated effectively.

There are two important issues with data stall cycles. First, the run-time occurrence of data cache misses is in general hard to predict at compile time. Compilers can attempt to schedule instructions according to their expected cache miss latency; such strategies, however, fail to capitalize on cache hits and can over-stress critical resources such as machine registers. Second, when a data stall arises, it is desirable to overlap the data stall cycles with other data stall cycles as well as computing cycles. This requires the ability to defer the execution of an instruction waiting for its data while allowing other load and compute instructions to proceed. Contemporary out-of-order designs rely on register renaming, dynamic scheduling, and large instruction windows to provide such concurrency.

Although these out-of-order execution mechanisms effectively hide data cache miss delays, they replicate, at great expense, much work done by the compiler. Regis-

*In American football, the *flea-flicker* offense tries to catch the defense off guard with the addition of a forward pass to a lateral pass play. Defenders covering the ball carrier thus miss the tackle and, hopefully, the ensuing play.

Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36 2003)
0-7695-2043-X/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY

ter 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 EPIC compilation and an in-order pipeline design while avoiding the penalty of cache miss stalls, this paper proposes a new microarchitectural organization employing two in-order sub-pipelines bridged by a first-in-first-out buffer (queue). The "advance" sub-pipeline, referred to as the *A-pipe*, executes all instructions speculatively *without stalling*. Instructions dispatching without all of their input operands ready, rather than incurring stalls, are suppressed, bypassing and writing specially marked non-results to their consumers and destinations. Other instructions execute normally. This propagation of non-results in the A-pipe to identify instructions affected by deferral is inspired by EPIC control speculation work [3]. The "backup" sub-pipeline, 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 in-order semantics in each.

This paper presents the design and evaluation of the *flea-flicker* two-pass pipelining model. We argue that two-pass pipelining effectively hides the latency of near cache accesses (such as hits in the L2 cache) and provides substantial performance benefit while preserving the most important characteristics of EPIC design. This argument is supported with simulations of SPEC95 and SPEC2000 benchmarks that characterize the prevalence and effects of the targeted latency events, demonstrate the effectiveness of the proposed model in achieving concurrent execution through these events, and inform the design decisions involved in building two-pass systems.

2 Motivation and case study

Before elaborating on the proposed microarchitectural extensions, it is useful to delineate the opportunities we hope to exploit with the help of a case study in a modern instruction-set architecture, the Intel Itanium Architecture. The Itanium Architecture defines a medium in which the compiler can produce an expression of program parallelism suitable for execution on a particular machine. This is achieved using a wide-word encoding technique in which groups of instructions intended by the compiler to issue together in a single processor cycle are delimited explicitly in the instruction encoding. Thus, the program encoding generated by the compiler is not a general *specification* of available parallelism, but rather a particular *implementation* within that constraint. In Itanium, these groups are separated by variably-positioned "stop bits." All instructions within 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. This design accommodates wide issue by reducing the complexity of the issue logic, but introduces the likelihood of "artificial"[1] dependences between instructions of unanticipated latency and instructions grouped with or subsequent to their consumers.

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% of execution cycles are consumed by data memory access-related stalls. Furthermore, depending on the benchmark, between 10% and 95% 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 two-pass design absorbs these events while allowing efficient exploitation of the compiler's generally good schedule.

Figure 1 shows an example from one of the most significant loops in the SPECint2000 benchmark with the most pronounced data cache problems, *181.mcf*. 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. To achieve such economy here, however, the compiler must 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). In general, the compiler cannot anticipate statically which loads will miss and when. A limited degree of dynamic execution could easily overcome this problem, but too high a degree, in ad-

[1] 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.

IEEE COMPUTER SOCIETY

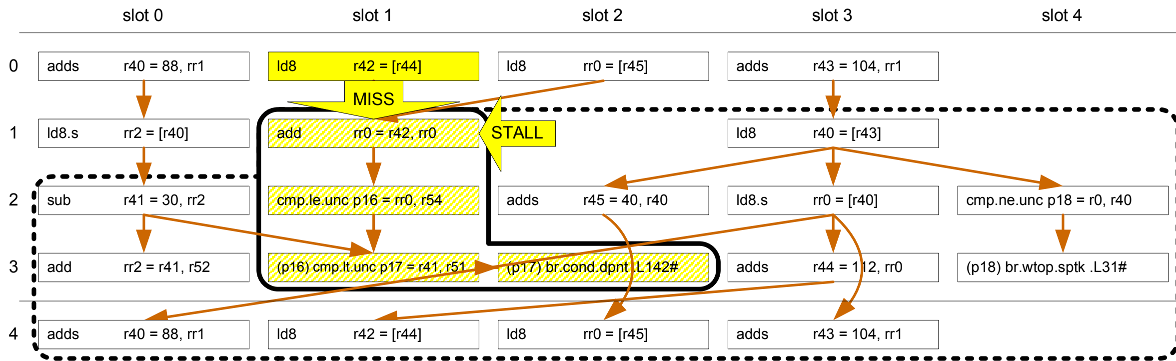

Figure 1: Cache miss stall and artificial dependences

dition 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, very substantially complicates the implementation of predicated execution, a fundamental feature of EPIC design [2].

Run-ahead pre-execution is one type of limited dynamic execution mechanism, in which instructions subsequent to the stalling instruction are executed during the stall to prepare microarchitectural state for more efficient execution of future instructions. Initial experiments using our baseline EPIC pipeline (see Section 4) and an idealized checkpointing-based pre-execution mechanism (synthesizing the ideas proposed by Dundas [5] and Mutlu [6]) revealed that a run-ahead approach could capitalize on near-misses, if it could be implemented in a way that did not impact the efficiency of the EPIC paradigm. We now present an implementation that, we believe, most efficiently provides the benefits of run-ahead in an EPIC context.

3 The two-pass pipeline scheme

The proposed two-pass pipeline scheme is designed to allow, during the memory stall cycles observed in traditional in-order pipelines, the productive processing of independent instructions. Figure 2(a) 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 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.

Figure 2(b) shows our proposed alternative. Here, when an instruction is found not to be ready in the dependence check, the processor, rather than stalling, marks the instruction and all *dependent* successors (as they arrive) 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, which does stall 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 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 concurrency.

3.1 Basic mode of operation

Figure 3 shows a potential design of the proposed mechanism as applied to an architecture similar to Intel's Itanium[2]. The reader will note marked similarities between the front end and architectural pipeline and the pipeline of

[2]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.

IEEE COMPUTER SOCIETY

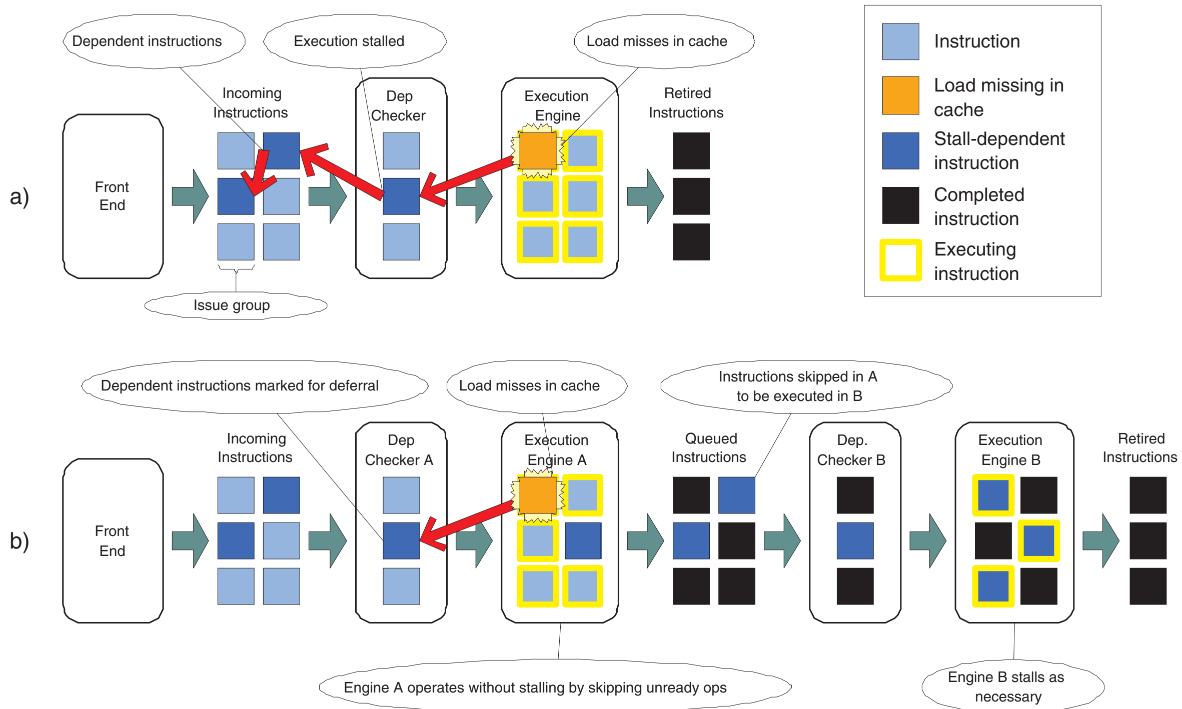

Figure 2: Snapshot of execution: (a) original EPIC processor; (b) two-pass pipeline.

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 Figure 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, rather than stall on unready instructions, the **REG** stage continues to process subsequent instructions by suppressing the deferred instructions and their data dependent successors. 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 Figure 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 Figure 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 suppressed 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 re-execute (in the B-pipe) instructions successfully pre-executed (or even pre-started) in the A-pipe is relieved. This has two effects: first, it reduces pressure on critical resources such as the memory subsystem interface; second, since these pre-executed instructions are free of input dependences when they arrive in the B-pipe, an opportunity for height reduction optimizations is created. In one simulated design, the

IEEE COMPUTER SOCIETY

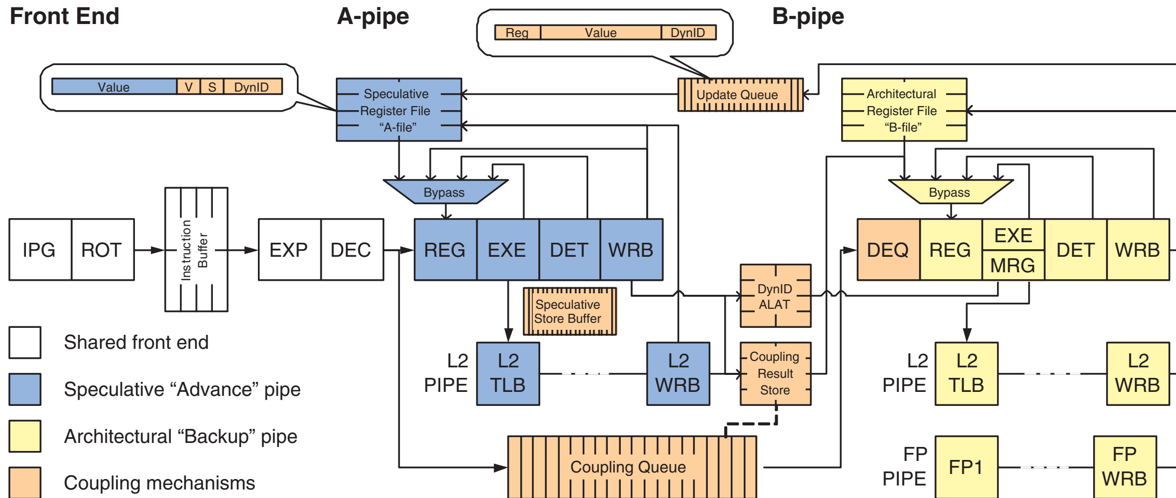

Figure 3: Two-pass pipeline design.

B-pipe dispatch logic re-groups (but does not reorder) instructions as they are dequeued, so that adjacent, independent instruction groups available at the end of the queue issue together, as machine resources allow. The experimental results (Section 4) indicate the potential benefit of *instruction regrouping* by removing stop bits rendered unnecessary by instruction pre-execution.

As an example of the concurrency exposed by the two-pass technique, Figure 4 shows four successive stages of execution of the code of Figure 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 Figure 4(a), a load issues in the A-pipe and misses in the first-level cache. In (b), a dependent 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 (c), 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 (d), two groups have retired from the A-pipe and re-execution has begun in the B-pipe. Many pre-executed 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 pre-executing 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 re-execute 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 pre-executed 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 in-

IEEE COMPUTER SOCIETY

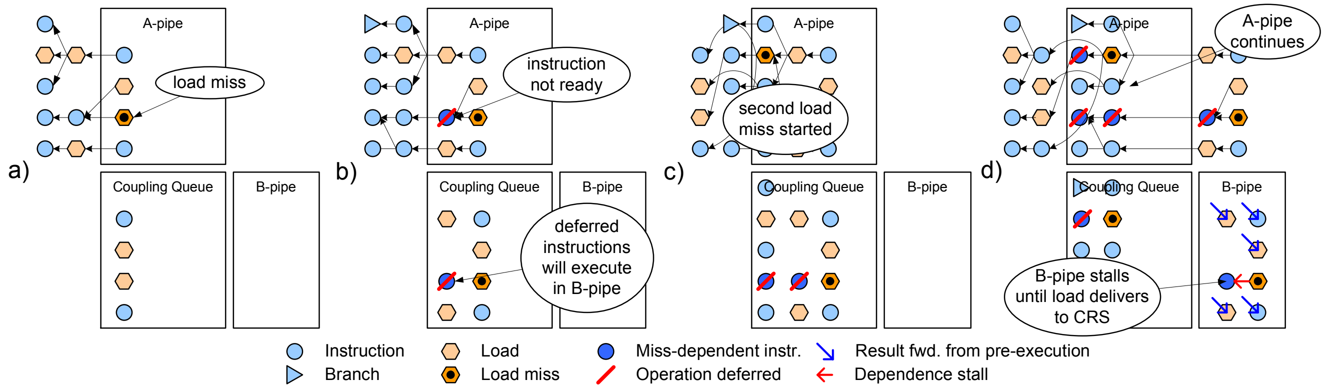

Figure 4: Applying two-pass pipelining to the previous *181.mcf* example.

struction 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 marked as such 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. 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 which end up having the same access address, but where program order is violated by the deferral of the first instruction to the B-pipe. Consider α to indicate an instruction which executes in the A-pipe and β one which executes in the B-pipe. Three dependence cases are of interest, as follows. Seemingly violated anti-dependences $\mathrm{ld}[addr]^\beta \xrightarrow{A} \mathrm{st}[addr]^\alpha$ and output dependences $\mathrm{st}[addr]^\beta \xrightarrow{O} \mathrm{st}[addr]^\alpha$ 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) [7, 8] 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 [9], 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. Here, 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, corrupted speculative state must be flushed (conservatively, all instructions subsequent to the load and all

0-7695-2043-X/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY

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 frequency 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 Maintaining the A-pipe success rate

Since an instruction gets only a single chance at pre-execution in the A-pipe, a deferred instruction and all its dataflow successors (until an update arrives from the B-pipe) will fully expose any stalls associated with the consumption of their results. In other words, the two-pass scheme provides one "second chance" to expose ILP; for these instructions, that chance has been lost. Figure 5 shows a dependence-graph representation of five issue groups to illustrate this limitation. After the cache miss caused by load instruction A, execution is not stalled with the arrival of consumer C. Rather, C is forwarded for execution to the B-pipe along with all its dataflow successors. Instruction D is thus allowed to begin execution in the A-pipe. The result is an overlapping of the miss latencies A→C and D→H. When the subsequent instruction F, executing in the B-pipe, also misses in cache, however, execution of the B-pipe is stalled at instruction J, resulting in the serialization of the latencies of instructions F and K. As a result of this limitation, one can consider the deferral of any unknown-latency instruction to the B-pipe to be something of a liability. Each pair of stalls in the deferred dependence chain of this instruction that are serialized by in-order semantics will be serialized in execution since there is no subsequent pipe to which to defer. Typically this is no worse than if we had a single pipeline; however, when such an instruction leads to the detection of a branch misprediction performance may suffer, since the two-pass scheme adds to the B-pipe branch misprediction penalty compared to a traditional in-order pipeline.

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. 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 Figure 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

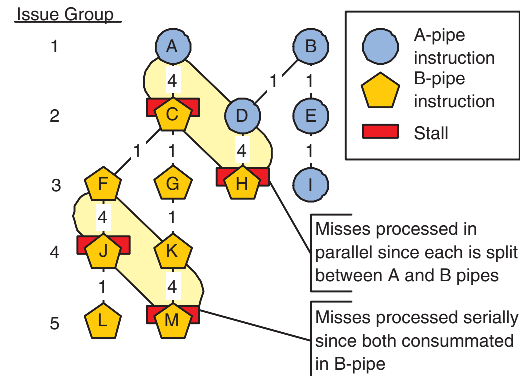

Figure 5: Limitation on overlap of latency events.

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.

Additionally, it may be useful to constrain the deferral of instructions by the A-pipe. If very little actual execution is occurring in the A-pipe (most instructions are being deferred), allowing the A-pipe to continue to run ahead may simply accumulate a large number of unexecuted instructions in the coupling queue—instructions which will cause non-overlappable stalls when executed. At some point, flushing instructions out of the queue and restarting the A-pipe issue after the B-pipe has cleared some of the backlog may be preferable to accumulating a long sequence of deferred instructions. This is a matter for future investigation.

3.6 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 Figure 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**, there is no need to stall the B-pipe 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,

IEEE COMPUTER SOCIETY

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 [10]. Various invalidation or A-file double-buffering strategies could also be applied.

3.7 Assessing the implementation cost

The actual execution pipelines of a typical, contemporary microprocessor consume only a small fraction of the chip's transistor count. For example, the Intel Itanium 2 integer pipeline and integer register file together consume an average of less than five percent of the total chip power and occupy less than two percent of the total chip area [11]. Even considering the relatively substantial additional overhead of things like pipeline control and clock distribution, the cost of two-pass pipelining appears reasonable. Additionally, our two-pass pipeline design requires relatively little interaction (e.g. no bypassing) between the two pipelines, resulting in minimal global complexity.

Nonetheless, it may not be desirable to replicate all functional units in both pipelines. As examples, the floating-point subpipeline would be a significant fraction of the replicated area, and data-cache ports might be prohibitively expensive to replicate. However, our design lends itself readily to partial replication. The only limitation is that a complete set of functional resources must be available to the B-pipe. Some units could be shared, with the two pipes arbitrating them, or, if the A-pipe does not have a particular type of unit available to it, instructions incapable of execution on the A-pipe can be marked as deferred and passed into the coupling queue. Of course, this can impact performance if instructions using non-replicated functional units occur frequently and are on paths leading to pipeline stalls. For our evaluation, we have assumed that each of the functional units have been replicated.

Compared to a full out-of-order implementation, two-pass pipelining avoids the significant area, power and complexity cost of instruction schedulers and register renaming hardware. While our system requires a second register file, out-of-order execution would also require additional registers (potentially more than twice the number of architectural registers) to support effective renaming. The buffer coupling the A-pipe with the B-pipe is a simple first-in/first-out queue that is likely less complex than a reorder buffer in an out-of-order architecture. Similarly the result queue for retired values from the B-pipe is also a simple first-in/first-out queue, and this feedback path adds only localized and latency-tolerant complexity. Finally, the ALAT hardware used in two-pass pipelining is similar in complexity to the content-addressable memories used by aggressive out-of-order implementations [9] to support the reordering of loads with stores to unknown addresses. While difficult for us to quantify exactly, the cost of two-pass pipelining appears, qualitatively, quite reasonable.

Table 1: Experimental machine configuration.

Feature	Parameters
Functional Units	8-issue, 5 ALU, 3 Memory, 3 FP, 3 Branch
Data model	ILP32 (integer, long, and pointer are 32 bits)
L1I Cache	2 cycle, 16KB, 4-way, 64B lines
L1D Cache	2 cycle, 16KB, 4-way, 64B lines
L2 Cache	5 cycles, 256KB, 8-way, 128B lines
L3 Cache	15 cycles, 1.5MB, 12-way, 128B lines
Max Outstanding Loads	16
Main memory	145 cycles
Branch Predictor	1024-entry gshare
Two-pass Coupling Queue	64 entry
Two-pass ALAT	perfect (no capacity conflicts)

4 Experiments

In order to examine the performance of the two-pass pipelined microarchitecture, we developed an execution model within our cycle-by-cycle full-pipeline simulation environment. The most relevant machine parameters from the simulated system are shown in Table 1. Our simulated design was based on pipeline length one stage longer than that of the Intel Itanium 2 architecture to model an achievable near-future EPIC microarchitecture. A cache hierarchy with latencies only slightly longer than those of the Itanium 2 was selected to provide a relatively conservative baseline memory hierarchy; a futuristic design with smaller low-level caches and longer latencies would further accentuate the demonstrated benefits.

For our evaluation, three benchmarks from SPECint95, six from SPECint2000, and one from SPECfp2000 were compiled using the IMPACT compiler. These are listed in Table 2 along with the simulated inputs and the lengths of these inputs in executed instructions. Each program was optimized using control-flow profiling information and pointer alias analysis. Optimizations applied included inlining, hyperblock formation, and instruction scheduling with control speculation. Due to the length of the SPEC-provided inputs, a mixture of SPEC test inputs and reduced SPEC reference inputs from the University of Minnesota (UMN) [12] were used for simulation.

Execution cycle counts for these benchmarks are shown

0-7695-2043-X/03 $17.00 © 2003 **IEEE**
IEEE COMPUTER SOCIETY

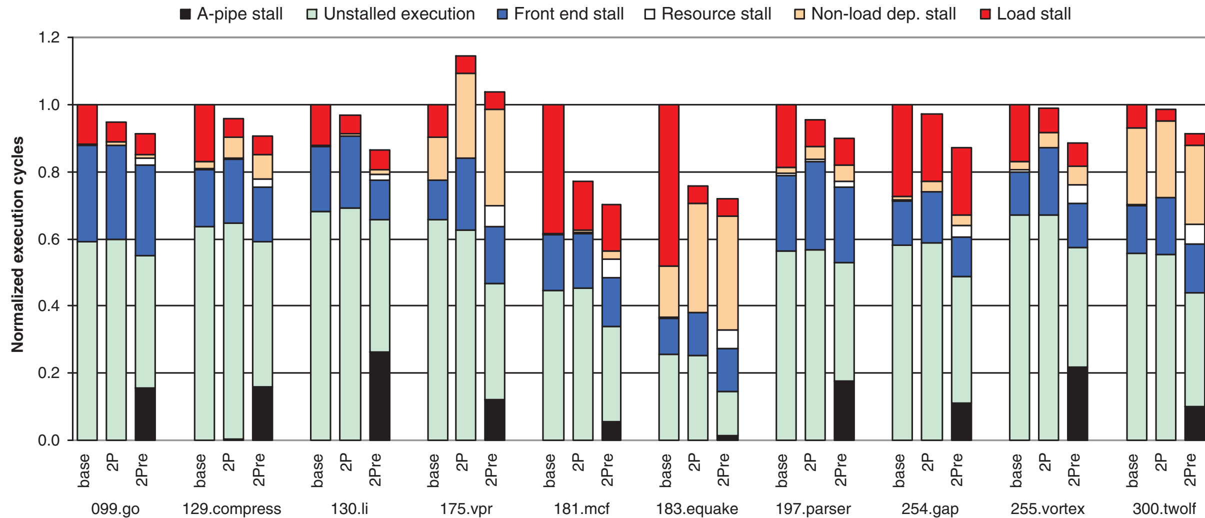

Figure 6: Normalized execution cycles; baseline (base), two-pass (2P) and with instruction regrouping (2Pre).

Table 2: Benchmarks and inputs used in experiments.

Benchmark	Inputs	Instructions
099.go	SPEC Train	453 M
129.compress	SPEC Train	13 M
130.li	SPEC Train	91 M
175.vpr	SPEC Test	1145 M
181.mcf	SPEC Test	132 M
183.equake	SPEC Test	477 M
197.parser	UMN mdred	409 M
254.gap	SPEC Test	645 M
255.vortex	UMN mdred	189 M
300.twolf	UMN smred	46 M

in Figure 6, normalized to the number of cycles in the baseline machine. The normalized execution cycles are presented for the baseline (**base**), two-pass pipelining (**2P**) and two-pass pipelining with instruction regrouping (**2Pre**). As described in Section 3.1, instruction regrouping allows the **2Pre** system to execute in a single cycle a set of instructions that spanned a stop bit in the original schedule. Because pre-executed instructions have no incoming, latency-bearing dependences, the regrouper removes the superfluous cycle break between two adjacent issue groups wherein instructions in the second group are no longer dependent upon those in the first because of A-pipe pre-execution.

In Figure 6, each cycle is classified into one of six categories of stall or unstalled execution conditions. For two-pass pipelining, these represent the condition of the B-pipe so that the architectural pipeline of the two-pass pipelined system is compared with that of the baseline. These cycle classes include stalls delaying the B-pipe because we require that the A-pipe always remain at least one cycle ahead of the B-pipe. (**A-pipe stall**), stalls on the front-end of the processor (**Front end stall**), stalls on oversubscribed resources (**Resource stall**), stalls on dependences (**Non-load dep. stall** and **Load stall**) and unstalled execution (**Unstalled execution**).

For each benchmark, a significant number of memory stall cycles is eliminated by two-pass pipelining. Generally, this improvement results in a reduction in executed cycles for the **2P** system. For example, *181.mcf* shows a 62% reduction in memory stall cycles and a 23% reduction in overall cycles. For other benchmarks like *300.twolf* the reduction in memory stall cycles is offset by an increase in additional cycles stalled in the front end. This is due to the effective lengthening of the pipeline observed by branch mispredictions resolved in the B-pipe. Although a small number of its load miss cycles are successfully hidden, *175.vpr* is the only benchmark to show a net loss of performance, due to store conflict flushes and dependence stalls. Its dependence stalls are caused by the deferral of 98% of its long-latency floating point instructions, in chains, to the B-pipe because the A-pipe does not stall for them to complete. It may therefore be advisable to allow the A-pipe to stall on *anticipable* latencies, since these latencies are effectively modeled by the compiler.

Figure 7 shows the distribution of the initiation of memory accesses to the A- and B-pipes. Each access is categorized by the level of the cache hierarchy from which it is serviced, and is scaled by the effective latency of an access to that level. For each benchmark, the majority of the access latency is initiated in the A-pipe, indicating that it is largely successful in pre-executing loads, with a smaller portion of accesses being deferred to the B-pipe. Most notable is the significant portion of the L3 cache misses in *183.equake* started in the A-pipe. The benefit of overlapping the handling of these cache misses is clearly reflected in the performance improvement for *183.equake*. *254.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. *254.gap* is also

IEEE COMPUTER SOCIETY

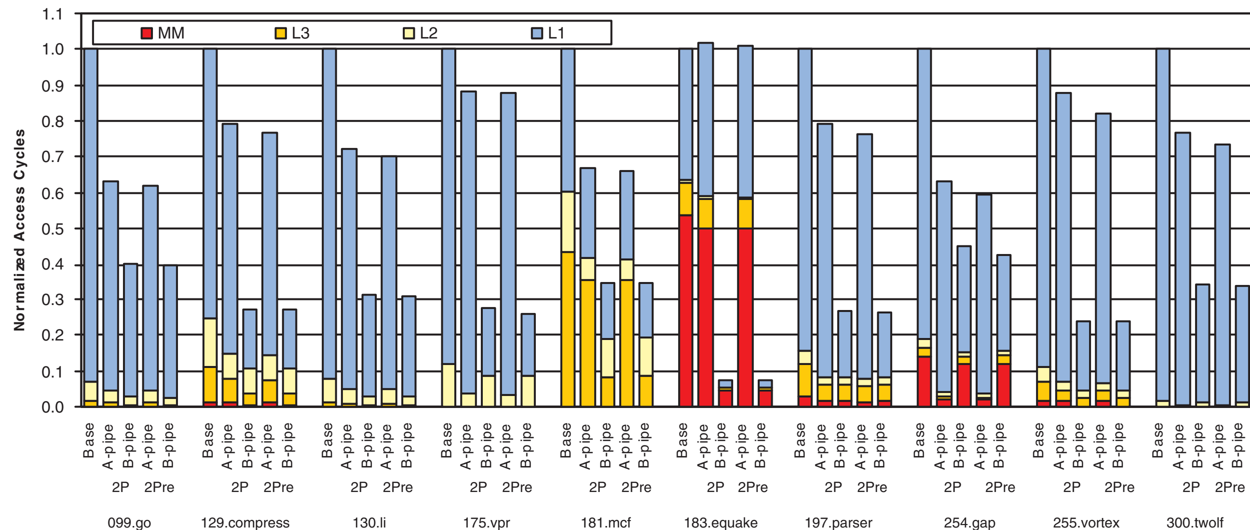

Figure 7: Distribution of initiated access cycles; baseline (base), two-pass (2P) and with instruction regrouping (2Pre).

an example of a benchmark showing a decrease in A-pipe loads with the addition of regrouping. Regrouping may allow the B-pipe to respond more rapidly to branch mispredictions, reducing the number of off-path loads futilely fetched, executed, and sent down the queue by the A-pipe.

We posit three modes of benefit from two-pass pipelining. First, 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. Second, long latency memory instructions which would have otherwise been blocked by preceding stalled instructions can be started early. This allows multiple long latency loads to be overlapped rather than processed sequentially. Both of these techniques reduce the number of cycles in which the processor reports being stalled on load misses, as demonstrated in Figure 6. When an application has poor cache locality, the benefit of overlapping long accesses dominates the benefit of hiding shorter ones (as in *183.equake*). For other benchmarks, like *129.compress*, the few long-latency memory accesses are distributed between the A-pipe and the B-pipe and the performance gain seen in **2P** for *129.compress* is likely due to the first source, the absorption of latencies from short but ubiquitous misses. Finally, regrouping instructions at the head of the B-pipe allows now-superfluous stop bits to be removed, increasing the degree of instruction-level parallelism available. This is clearly shown in Figure 6, with **2Pre** achieving an average speedup of 1.08 over **2P**.

Two-pass execution has the potential to sacrifice performance in two 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 32% 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 during the redirection of the A-pipe as long as the coupling queue has instructions remaining. 68% of branch mispredictions remain to be processed in the B-pipe.

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 97% of all load accesses initiated in the A-pipe while a deferred store is in the queue are free of store conflicts. Only 1.6% 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 Figure 6 show by the reasonable size of the **Front end stall** segment that neither substantially erodes the performance gained from two-pass pipelining.

Finally, continued successful pre-execution requires that committed results in the B-pipe be fed back into the A-pipe to prevent the deferral of ever-greater numbers of instructions. As this inter-pipe communication may be difficult to implement in a single cycle, we evaluated the effect of latency on this update path for three of the benchmarks, as shown in Figure 8. As shown by the increase in deferred instructions when this updating is eliminated (**inf**), it is clear such a mechanism is beneficial. For *181.mcf*, eliminating the feedback increases by 16% the number of instructions deferred. Loss of pre-execution potential for these instructions increases *mcf's* runtime by 5.5%. On the other hand, Figure 8 also shows that this update path is tolerant of moderate additional latency, especially up to four clock cycles.

0-7695-2043-X/03 $17.00 © 2003 IEEE

IEEE COMPUTER SOCIETY

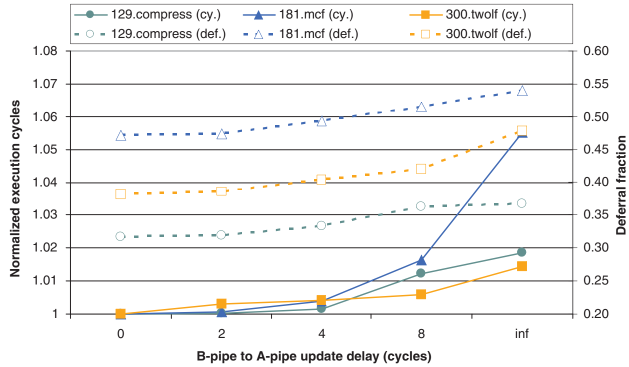

Figure 8: Effect of delay in feeding committed results in the B-pipe to the A-pipe.

5 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 [5] and Mutlu [6] both propose run-ahead schemes relying on checkpointing and repair. Mutlu presents a run-ahead implementation specifically targeting long-latency misses in out-of-order machines. The technique attempts to accommodate such misses efficiently without over-stressing out-of-order instruction scheduling resources. In a single-issue, short-pipeline, in-order machine, Dundas examines run-ahead execution in a special mode during any L1 cache miss. In Dundas' 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' 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 [13] and master/slave speculative parallelization [14] attempt to exploit additional instruction-level parallelism by selecting program threads for pre-execution. 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 share 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 [15] 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 [16] 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.* [17, 18] proposed software-based speculative precomputation and prefetching targeted to particular delinquent loads. Annavaram [19] proposed a dynamic mechanism to generate prefetching microthreads for pre-execution. 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.

6 Conclusion and future work

This paper presents the two-pass pipeline organization, a mechanism that counteracts in-order stalls on unanticipated cache latencies while preserving the benefits of traditional in-order pipelines. We present the basic organization and outline the critical design issues encountered in its development. This novel microarchitecture achieves many of the objectives of out-of-order execution models without dynamic scheduling and renaming. We show with initial simulation results that executing ahead, beyond stalled instructions, in one "advance" pipeline allows a significant portion of cache miss latency to be tolerated. This is shown to result

IEEE COMPUTER SOCIETY

in significant performance improvement for benchmarks that spend much of their execution time stalled on memory accesses. Finally, the increase in effective pipeline length for the execution of deferred instructions in a "backup" pipeline is shown not to overwhelmingly degrade performance with flush cycles.

In our detailed results, we show how our initial design addresses these issues. Specifically, we have shown that the bulk of load instructions are initiated in the A-pipe, allowing short-latency misses to be absorbed and long latency misses that would have been serialized by stalls in a traditional organization to be resolved concurrently. Additionally the feedback of values from the B-pipe to the A-pipe is shown to tolerate a reasonable amount of latency.

We expect the benefit of our proposed pipeline organization to be even more pronounced for applications that stress the cache hierarchy, and that the benefit of the proposed pipeline organization will further increase for future processors which are bound to be more distant from substantial cache storage. The two-pass design space supports many research opportunities, including the study of mechanisms to moderate the issue of the A-pipe in cases where most instructions are being deferred to the B-pipe.

Acknowledgments

This work was supported by the MARCO/DARPA Center for Circuits, Systems and Software under contract 2001-CT-888 and the National Science Foundation Information Technology Research program under contract number 0086096. We thank John Crawford, John Shen, Chris Newburn and Matthew Merten at Intel Corporation and Jim McCormick at Hewlett-Packard for their generous and insightful feedback. We also thank Jason Park for his work on our simulation framework 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," in *Proceedings of the 25th Annual International Symposium on 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," in *Proceedings of the Eighth International Symposium on 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 Transactions on Computer Systems (TOCS)*, vol. 11, no. 4, pp. 376–408, 1993.

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

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

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

[7] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu, "Dynamic memory disambiguation using the Memory Conflict Buffer," in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 183–193, Oct. 1994.

[8] R. Zahir, J. Ross, D. Morris, and D. Hess, "OS and compiler considerations in the design of the IA-64 Architecture," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 213–222, Oct. 2000.

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

[10] W. W. Hwu and Y. N. Patt, "Checkpoint repair for out-of-order execution machines," in *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pp. 18–26, July 1987.

[11] 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 Journal of Solid-State Circuits*, vol. 37, Nov 2002.

[12] AJ. KleinOsowski and D. J. Lilja, "MinneSPEC: A new SPEC 2000 benchmark workload for simulation-based computer architecture research," *Computer Architecture Letters*, vol. 1, May 2002.

[13] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, "A study of slipstream processors," in *Proceedings of the 33rd Annual International Symposium on Microarchitecture*, pp. 269–280, Nov. 2000.

[14] C. Zilles and G. Sohi, "Master/slave speculative parallelization," in *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pp. 85–96, Nov. 2002.

[15] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt, "Simultaneous subordinate microthreading (SSMT)," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp. 186–195, July 1999.

[16] J. R. Goodman, J. Hsieh, K. Liou, A. R. Pleszkun, P. Schechter, and H. C. Young, "PIPE: A VLSI decoupled architecture," in *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pp. 20–27, July 1985.

[17] 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," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pp. 14–25, July 2001.

[18] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, "Dynamic speculative precomputation," in *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pp. 306–317, Nov. 2001.

[19] M. Annavaram, J. M. Patel, and E. S. Davidson, "Data prefetching by dependence graph precomputation," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pp. 52–61, July 2001.

0-7695-2043-X/03 $17.00 © 2003 **IEEE**

IEEE COMPUTER SOCIETY