

iCFP: Tolerating All-Level Cache Misses in In-Order Processors

Andrew Hilton, Santosh Nagarakatte, and Amir Roth

Department of Computer and Information Science, University of Pennsylvania

{adhilton, santoshn, amir}@cis.upenn.edu

Abstract

Growing concerns about power have revived interest in in-order pipelines. In-order pipelines sacrifice single-thread performance. Specifically, they do not allow execution to flow freely around data cache misses. As a result, they have difficulties overlapping independent misses with one another.

Previously proposed techniques like Runahead execution and Multipass pipelining have attacked this problem. In this paper, we go a step further and introduce iCFP (in-order Continual Flow Pipeline), an adaptation of the CFP concept to an in-order processor. When iCFP encounters a primary data cache or L2 miss, it checkpoints the register file and transitions into an “advance” execution mode. Miss-independent instructions execute as usual and even update register state. Miss-dependent instructions are diverted into a slice buffer, un-blocking the pipeline latches. When the miss returns, iCFP “rallies” and executes the contents of the slice buffer, merging miss-dependent state with miss-independent state along the way. An enhanced register dependence tracking scheme and a novel store buffer design facilitate the merging process.

Cycle-level simulations show that iCFP out-performs Runahead, Multipass, and SLTP, another non-blocking in-order pipeline design.

1. Introduction

Growing concerns about power have revived interest in in-order processors. Certainly designs which target throughput rather than single-thread performance, like Sun’s UltraSPARC T1 “Niagara” [11], favor larger numbers of smaller in-order cores over fewer, larger out-of-order cores. More recently, even high-performance chips like IBM’s POWER6 [12] have abandoned out-of-order execution. In-order pipelines have area and power efficiency advantages, but sacrifice single-thread performance to achieve them. Specifically, they allow only limited execution around data cache misses—the pipeline stalls at the first miss-dependent instruction—and have difficulties overlapping independent misses

with each other. Ironically, this relative disadvantage diminishes in the presence of last-level cache misses. Here, both types of processors are similarly ineffective.

Continual Flow Pipelining (CFP) [24] exposes instruction- and memory- level parallelism (ILP and MLP) in the presence of last-level cache misses. On a miss, the scheduler drains the load and its dependent instructions from the window and into a slice buffer. These instructions release their physical registers and issue queue entries, freeing them for younger instructions. This “non-blocking” behavior allows the window to scale virtually to large sizes. When the miss returns, the contents of the slice buffer re-dispatch into the window, re-acquire issue queue entries and physical registers, and execute. The key to this enterprise is decoupling deferred slices from the rest of the program by buffering miss-independent inputs along with the slice.

CFP was introduced in the context of out-of-order checkpoint-based (CPR) processors [1], but its authors observed that it is a general concept that is applicable to many micro-architectures [17, 20]. In this paper, we adapt CFP to an in-order pipeline. An in-order pipeline does not have an issue queue or a physical register file—here CFP unblocks the pipeline latches themselves. Our design is called *iCFP* (*in-order Continual Flow Pipeline*) and it tolerates misses in the data cache, the last-level cache and every cache in between.

iCFP is not the first implementation of CFP in an in-order pipeline. SLTP (Simple Latency Tolerant Processor) [17] is a similar contemporaneous proposal. Like SLTP, iCFP un-blocks the pipeline on cache misses, drains miss-dependent instructions—along with their miss-independent side inputs—into a slice buffer and then re-executes only the slice when the miss returns. Re-executing only the miss-dependent slice gives SLTP and iCFP a performance advantage over techniques like Runahead execution [8] and “flea-flicker” Multipass pipelining [3], which un-block the pipeline on a miss but then re-process all post-miss instructions. iCFP has an additional advantage over SLTP. In SLTP, the pipeline is

non-blocking only in the shadow of misses; miss slice re-execution is blocking. This limits performance in dependent miss scenarios. In contrast, iCFP is non-blocking under all misses. iCFP may make multiple passes over the contents of the slice buffer, with each pass executing fewer instructions. Non-blocking also enables interleaving of slice re-execution with the execution of new instructions at the “tail” of the program. Our experiments show that these features contribute significantly to performance.

Supporting non-blocking slice execution requires two innovative mechanisms. The first is a register dependence tracking scheme that supports both multiple slice re-executions and incremental updates to primary register state. The second is a scheme that supports non-speculative store-load forwarding for miss independent stores and loads under a cache miss and for miss-dependent stores and loads during multiple slice re-executions. For the first, we use a second register file—which is available on a multi-threaded processor—as scratch space for re-executing slices. We use instruction sequence numbering to gate updates to primary register state. For the second, we describe a novel but simple store buffer design that supports forwarding using “chained” iterative access rather than associative search.

2. Motivation and Some Examples

iCFP implements an in-order pipeline that combines two important features. First, it supports non-blocking uniformly under all misses. This allows it to “advance” past primary misses (encountered while no other miss is pending), secondary misses (encountered while the primary miss is pending), and dependent misses (encountered while re-executing the forward slice of a primary or secondary miss). Second, while advancing under any miss, iCFP can “commit” miss-independent instructions. When any miss returns, iCFP “rallies” by re-executing only the instructions that depend on it.

This combination of features allows iCFP to effectively deal with different patterns of misses, whereas micro-architectures that implement only one of these features have limited effectiveness when encountering certain miss patterns. For instance, SLTP [17] commits miss-independent instructions but blocks when re-executing miss slices. As a result, it provides limited benefit in dependent-miss scenarios. Runahead execution [8] implements general non-blocking but must re-execute miss-independent instructions. This limits its effectiveness in general, but especially in scenarios in which long-latency primary L2 misses are followed by shorter secondary data cache misses.

Figure 1 uses abstract instruction sequences to illus-

trate the actions of a vanilla in-order pipeline, Runahead execution (RA), SLTP, and iCFP in different scenarios. In the examples, instructions are boxed letters, cache misses are shaded, and data dependences are arrows. In a vanilla pipeline, misses result in pipeline stalls (thick horizontal lines). In RA, SLTP, and iCFP, they trigger advance execution. Miss-dependent advance instructions—also known as “poisoned” instructions—are shown as lower-case letters. For RA, SLTP, and iCFP, advance and rally execution are split, with advance execution on top.

Lone L2 miss. Figure 1a shows a lone L2 miss (A) with a single dependent instruction (B). In this situation, RA provides no benefit. SLTP and iCFP do because they can commit miss-independent advance instructions C–F, and re-execute only the miss forward slice (A–B).

Independent L2 misses. Figure 1b shows independent L2 misses, A and E. In a vanilla pipeline, these misses are serialized. However, RA, SLTP, and iCFP can all overlap these misses by advancing under miss A. Note, SLTP slice re-execution is blocking and so it must wait until E completes before finishing the rally. In contrast, iCFP can interleave execution at the “tail” of the program (G–H) with slice re-execution.

Dependent L2 misses. Figure 1c shows a dependent-miss scenario—E depends on A. RA is ineffective here. SLTP provides a small benefit because it can commit instructions C and D in the shadow of miss A. However, the fact that it has blocking rallies prevents it from committing additional instructions under miss E. iCFP is not limited in this way.

Independent chains of dependent L2 misses. Figure 1d shows four misses, A, B, E and F with pairwise dependences between them—B depends on A and F on E. Assume L2 miss latency is long enough such that advance execution under miss A can execute E before A returns. RA is effective, overlapping E with A and F with B, respectively. Despite being able to commit miss-independent advance instructions, SLTP is less effective than RA. Although it can overlap A with E during advance mode, its blocking rallies force it to serialize B and F. Again, iCFP does not have this limitation.

Secondary data cache misses. Earlier, we mentioned that the inability to reuse miss-independent instructions limits RA in certain miss scenarios. Figures 1e and 1f illustrate. The scenarios of interest involve secondary data cache misses under a primary L2 miss. In these situations, RA advance execution is faced with a choice. On one hand, it can wait for the miss to return. We call this option D\$-blocking (D\$b), and it is the right choice if there are future misses that de-

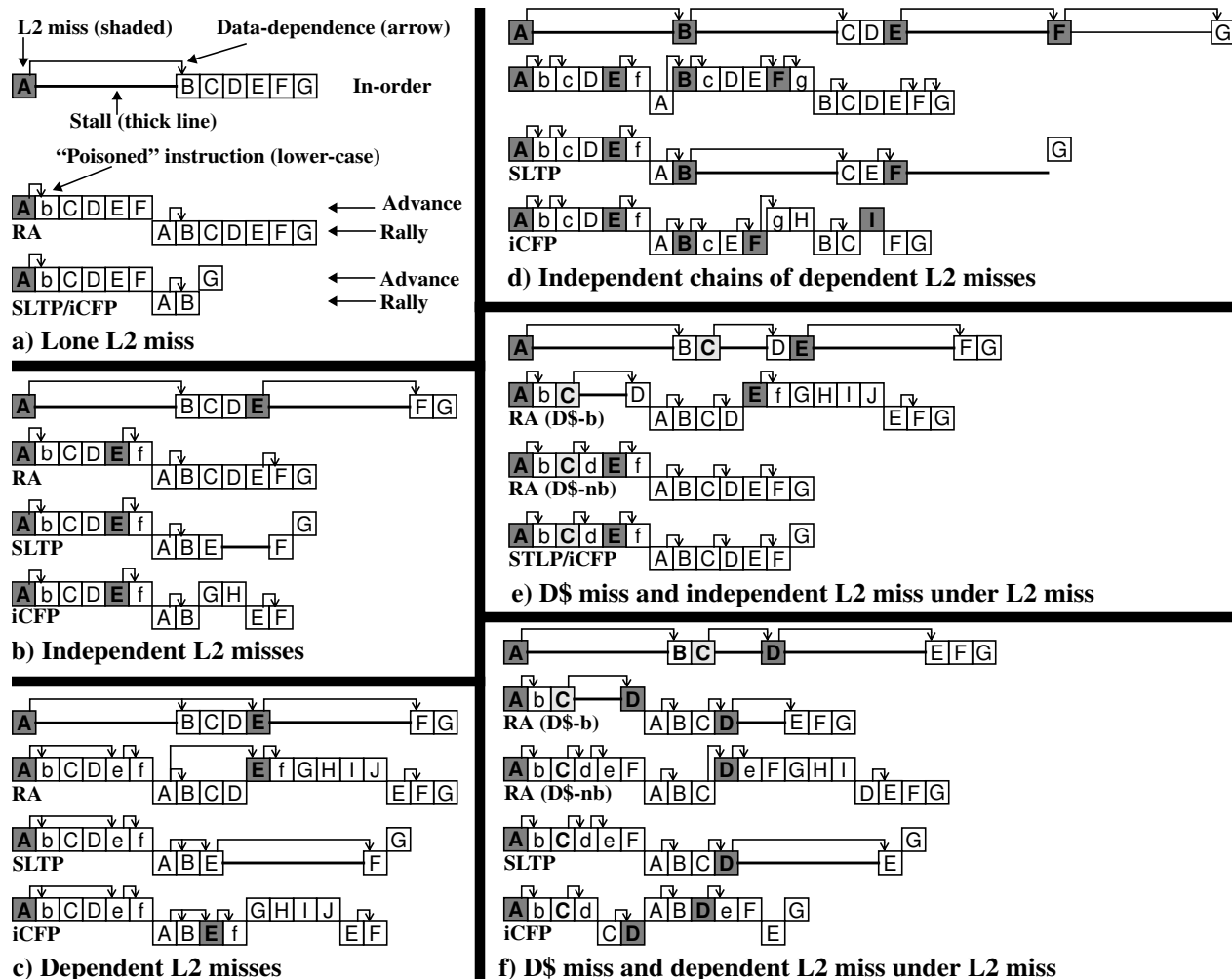


Figure 1. In-order, RA (Runahead), SLTP, and iCFP under different miss scenarios

pend on the data cache miss (as in Figure 1f, with D depending on C). Blocking is a poor choice if there are future misses that are independent of the data cache miss because waiting for the data cache miss will delay those misses. This is the case in Figure 1e, where waiting for C prevents overlapping D with A. Alternatively, RA can “poison” the output data cache miss and proceed immediately, this is the D\$-non-blocking (D\$-nb) option. Non-blocking is right if there are future independent misses (Figure 1e) and wrong if there are future dependent misses (Figure 1f). We note that only RA is faced with this particular dilemma. iCFP can confidently poison the secondary data cache miss because it can return to it immediately when the miss returns. With some caveats, SLTP can do the same. But RA—because it doesn’t buffer and decouple miss-dependent instructions—can only return to the primary

“outer” miss. Our experiments show that most benchmarks prefer D\$-blocking.

3. iCFP: In-order Continual Flow Pipeline

iCFP requires a set of simple extensions to an in-order processor. These include: i) a mechanism for checkpointing and restoring the contents of the register file, ii) per register “poison” bits and sequence numbers for tracking miss-dependent instructions, iii) a FIFO for buffering miss-dependent instructions and their side inputs, iv) a “scratch” register file for re-executing miss-dependent slices, v) a mechanism that supports correct store-load forwarding during both advance and rally execution, and vi) a mechanism for detecting shared-memory conflicts at checkpoint granularity.

Figure 2 shows a simplified structural diagram of an iCFP pipeline (parts c and d) and similar diagrams for a vanilla in-order pipeline (part a) and a Runahead

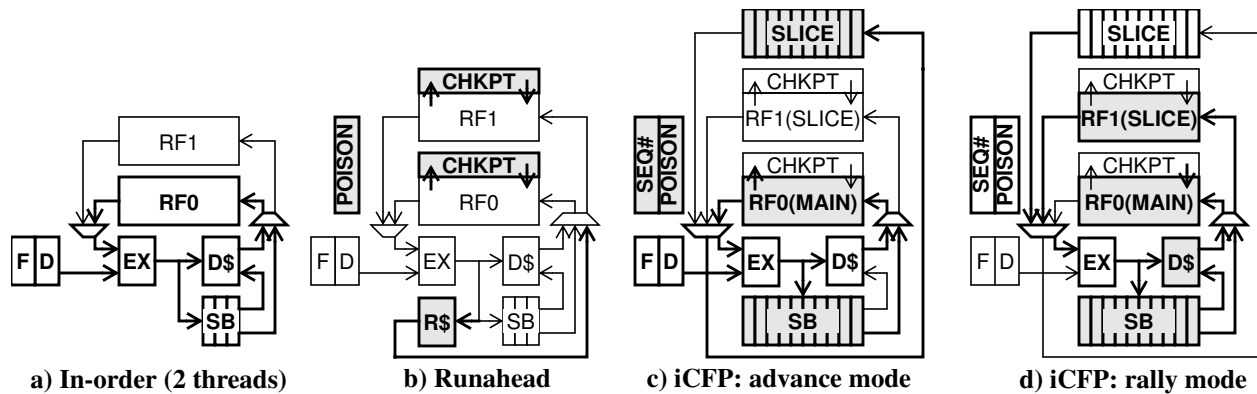


Figure 2. In-order, Runahead, and iCFP structural diagrams

pipeline (part b) [6, 8]. Some of the mechanisms required by iCFP are also required by Runahead. These include the register checkpointing mechanism and the poison bits. The kind of checkpointing Runahead and iCFP require—a single checkpoint that supports only create and restore operations—can be implemented efficiently using “shadow” bitcells [9].

iCFP needs an additional FIFO to buffer miss-dependent instruction slices (SLICE in parts c and d), and an additional register file for implementing register communication during slice re-execution. Many in-order processors support multi-threading and so effectively contain multiple register files [11, 12]. iCFP simply borrows a register file for this purpose when single-thread performance trumps multi-thread throughput.

iCFP uses simple yet novel components to provide store-load forwarding and inter-thread memory ordering violation detection. For forwarding, it uses an indexed store buffer with a novel access method called *address-hash chaining* (Section 3.2). For multi-processor safety, it uses a signature scheme (Section 3.3).

3.1. Advance and Rally

Figures 2c and 2d show iCFP’s active components during advance and rally execution, respectively. Components with thick outlines are active. Shaded components are actively updated.

Advance execution. iCFP advance execution resembles that of Runahead. On a cache miss, the processor checkpoints the register file and “poisons” the output register of the load. Advance execution propagates this poison through data dependences using poison bits associated with each register and each store buffer entry (Section 3.2). Miss-independent instructions (instructions with no poisoned inputs) write their values into the register file. Non-poisoned branches are resolved as usual, triggering pipeline flushes on mis-predictions.

Miss-dependent instructions (instructions with at least one poisoned input) do not execute. They drain to the slice buffer along with their non-poisoned input (if any).

In iCFP, each register is associated not only with a poison bit, but also with a *last-writer sequence number*. An instruction’s sequence number is its distance from the checkpoint and sequence numbers determine relative instruction age. At writeback, all advance instructions—poisoned or not—update last-writer field of their destination register with their own sequence number. The sequence number field is used to prevent write-after-write hazards during rallies.

Rally execution. iCFP advance mode resembles that of Runahead, but its rally mode is different. In rally mode, iCFP re-injects the miss-dependent instructions from the slice buffer into the pipeline. These instructions obtain their miss-independent inputs from their slice buffer entries. They obtain inputs generated by older miss-dependent instructions either via the bypass network or the “scratch” register file which is used as temporary storage during rallies. A re-executing miss-dependent instruction updates the main register file only if the main register’s last-writer sequence number matches its own. If the register is tagged with a larger sequence number (*i.e.*, one from a younger instruction), the write is suppressed to avoid a write-after-write violation.

Like Multipass [3], iCFP may make multiple rally passes over the slice buffer, initiating a pass every time a pending miss returns. Each rally pass processes fewer instructions, until the slice is completely processed. During a rally, not all loads in the slice buffer may have returned—in fact, dependent loads may just have issued for the first time and initiated misses. iCFP does not stall the rally to wait for these loads to complete. The scratch register file also has associated poison bits and when a rally begins, these are cleared. During a rally, any still-

Instructions (from fetch)			seq	p	RF0 val seq p		
ld	[r1{x40}]	→ r3{-}	0	1	r1	x44	4 0
ld	[r2{xC0}]	→ r4{2}	1	0	r2	xC4	5 0
mul	r3{-}	, r4{2} → r4{-}	2	1	r3	3	6 0
st	r4{-}	→ [r1{x40}]	3	1	r4	-	8 1
addi	r1{x40}	, 4 → r1{x44}	4	0	RF1 val seq p		
addi	r2{xC0}	, 4 → r2{xC4}	5	0	r1	-	- 0
ld	[r1{x44}]	→ r3{3}	6	0	r2	-	- 0
ld	[r2{xC4}]	→ r4{-}	7	1	r3	-	- 0
mul	r3{3}	, r4{-} → r4{-}	8	1	r4	-	- 0
st	r4{-}	→ [r1{x44}]	9	1			

a) advance execution

Instructions (from slice buffer)			seq	p	RF0 val seq p		
ld	[SL{x40}]	→ r3{9}	0	0	r1	x44	4 0
mul	r3{9}	, SL{2} → r4{18}	2	0	r2	xC4	5 0
st	r4{18}	→ [SL{x40}]	3	0	r3	3	6 0
ld	[SL{xC4}]	→ r4{-}	7	1	r4	-	8 1
mul	SL{3}	, r4{-} → r4{-}	8	1	RF1 val seq p		
st	r4{-}	→ [SL{x44}]	9	1	r1	-	- 0

b) first rally

Instructions (from slice buffer)			seq	p	RF0 val seq p		
ld	[SL{xC4}]	→ r4{4}	7	0	r1	x44	4 0
mul	SL{3}	, r4{4} → r4{12}	8	0	r2	xC4	5 0
st	r4{12}	→ [SL{x44}]	9	0	r3	3	6 0

c) second rally

Instructions (from slice buffer)			seq	p	RF0 val seq p		
ld	[SL{xC4}]	→ r4{4}	7	0	r1	x44	4 0
mul	SL{3}	, r4{4} → r4{12}	8	0	r2	xC4	5 0
st	r4{12}	→ [SL{x44}]	9	0	r3	3	6 0

Figure 3. iCFP working example

pending loads are poisoned in the scratch register file and “re-activated” in their existing slice buffer slots. Effectively, rallies themselves perform advance execution.

Working example. Figure 3 shows an example of advance and rally execution in a parallel miss scenario. There is one advance pass (part a) and two subsequent rallies (parts b and c). Each pass shows the instruction stream, which comes from fetch during advance execution and from the slice buffer during rallies. Each instruction is tagged with a poison bit (p) and a sequence number from the checkpoint (seq). Each pass also shows the contents of the main and slice register files (RF0 and RF1, respectively). Each register is tagged with a poison bit and a last-writer sequence number.

The advance pass slices out the six shaded instructions, which form two dependence chains. At the end of the advance pass, main register r4 is poisoned and tagged with the sequence number of its last writer (8).

The first rally is triggered when the first load miss (sequence number 0) returns. The second load (sequence number 7) has not returned and so the instructions that depend on it (shaded) are re-poisoned and re-activated in the slice buffer. The slice executes using RF1 as scratch space. Notice, miss-independent inputs come from the slice buffer (SL) rather than the register file (RF0). This

example demonstrates the need for a scratch register file to execute slices. Rally instructions (sequence numbers 0 and 2) cannot write r3 and r4 into the main register file because these are already over-written by logically younger instructions (sequence numbers 6 and 8). The sequence numbering scheme helps avoid these write-after-write hazards.

The return of the second load miss (sequence number 7) triggers the second rally. This time, main register r4 is tagged with the sequence number of a rally instruction (8), and so the rally updates the main register file and unpoisons the register.

When the second rally completes, the slice buffer is empty and the main register file (RF0) is poison-free. Conventional in-order execution resumes. The next miss will trigger a transition to advance mode.

Multithreaded rally. Slices are dependence chains and unlikely to have an internal parallelism of greater than one. Even if they did, it is not likely that an in-order processor could exploit this parallelism. As a result, it makes little sense to allocate rally bandwidth greater than one instruction per cycle, even if the processor is capable of more.

For maximum throughput, iCFP executes rally instructions and tail instructions in multithreaded fashion, with rally instructions given priority. Such multithreading is possible because rally instructions are effectively decoupled from the rest of the program by virtue of having captured their miss-independent inputs during entry into the slice buffer. Slice instructions are identified explicitly so the pipeline can ignore dependences between slice and tail instructions. Multithreaded rallying requires the guarantee that as slices are being processed from the head of the slice buffer, they can continue to be properly extended at the tail. iCFP’s gated updates of main register file poison bits provide this guarantee.

3.2. Store-Load Forwarding

Advance instructions can write to the main register file because it is backed by a checkpoint. The data cache is not backed by a checkpoint and so advance stores cannot write to it. We also don’t assume data cache support for speculative “transactional” writes or write logging and rollback [15]. iCFP needs a mechanism to buffer advance stores so that they drain to the cache in program order, forward to younger miss-independent loads during advance execution, and forward to younger miss-dependent loads during rallies.

Runahead execution uses a Runahead cache (R\$ in Figure 2b) to support forwarding from advance stores to miss-independent advance loads in a scalable way [16]. But iCFP requires a more robust mechanism. A Run-

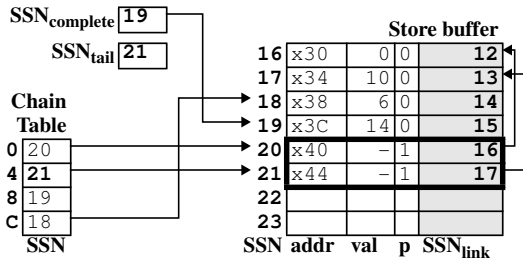


Figure 4. Address-hash chaining

head cache supports only “best-effort” forwarding because relevant stores may have been evicted from it. It also does not support program order data cache writes. Runahead does not need these features because it re-executes *all* advance instructions anyway. However, iCFP re-executes only miss-dependent advance instructions. It does not re-execute miss-independent stores and so its mechanism must not evict them. It also doesn’t re-execute miss-independent loads and so its mechanism must guarantee correct forwarding for them during advance execution.

A simple mechanism that provides these features is an associatively-searched store buffer, a structure already found in many in-order processors (for ISAs whose memory models permit). However, traditional store buffers have relatively few entries as they are primarily used to tolerate store miss latency and improve data cache bandwidth utilization. iCFP advance mode may last for many cycles, so the number of advance stores may be large. Large associative structures are slow, and area and power inefficient. iCFP uses a large store buffer that supports forwarding without associative search, using a technique we call *address-hash chaining*.

Address-hash chaining. Figure 4 illustrates address-hash chaining using the code example from Figure 3. The figure shows an 8-entry store buffer which contains two valid stores (in thick outline). Address-hash chaining uses the SSN (store sequence number) dynamic store naming scheme [21]. SSNs are extended store buffer indices that can also name stores that are already in the cache. A store’s store buffer index is the low order bits of its SSN. In the example, the two stores have SSNs 20 and 21 and are at store buffer indices 4 and 5, respectively. A global counter $SSN_{complete}$ tracks the SSN of the youngest store to write to the cache (here 19).

Each store buffer entry contains an address, value, poison bit, and an explicit SSN (SSN_{link}) which is not its own. The store buffer is coupled with a small, address-indexed table called the *chain table* which maps hashed addresses (*e.g.*, low-order address bits) to SSNs. Each chain-table entry contains the SSN of the youngest store

whose address hashes into that entry. For instance, the entry for low-order address bits 4 points to the store to address x44 (SSN 21). The SSN_{link} in each store buffer entry points to the next youngest store that has the same address-hash. For example, the SSN_{link} for SSN 21 (address x44) is 17; the store at SSN 17 writes address x34. Essentially, all entries in the store buffer are chained by hashed address with the chain table providing the “root set”. SSNs older than $SSN_{complete}$ correspond to stores that are already in the cache, and act as chain-terminating “null pointers”.

Loads forward from the store buffer by following the chain that starts at the chain table entry corresponding to their address. If a load finds a store with a matching address, it forwards from it. If the store is marked as having a poisoned data input, the poison propagates to the load, which then drains to the slice buffer. If the chain terminates before the load finds a store with a matching address, then the load gets its value from the data cache. With reasonable chain table sizes (*e.g.*, 64 entries), average chain length can be kept short and average load latency low. Our experiments show that the average number of excess store buffer hops per load—the first store buffer access is “free” because it is performed in parallel with data cache access—is less than 0.5 for all benchmarks and less than 0.05 for most. Address-hash chaining does produce variable load latency, but this is easier to manage in in-order processors, which do not use speculative wakeup.

Address-hash chaining supports forwarding to miss-dependent loads during rallies. Because the chain table corresponds to the tail of the instruction stream, it may contain pointers to stores that are younger than miss-dependent loads. This is not a problem. Re-executing miss-dependent loads simply follow the chain until they encounter stores that are older than they are.

Address-hash chaining must stall in one specific situation—a miss-dependent store with a poisoned address. Poison-address stores are relatively rare and are typically associated with pointer chasing. An address-poisoned store cannot be properly chained into the store buffer and proceeding past it removes all forwarding guarantees for younger advance loads. When iCFP encounters a poison-address store, it can either stall or transition to a “simple runahead” mode that does not commit miss-independent results.

3.3. Multiprocessor Safety

iCFP uses non-speculative same-thread store-load forwarding and does not suffer from same-thread memory ordering violations. However, being checkpoint-based makes iCFP’s loads vulnerable to stores from

other threads. iCFP must snoop these loads efficiently.

A large associatively-searched load queue is one option, but iCFP uses a cheaper scheme based on signatures [4]. iCFP maintains a single local signature. Loads which get their values from the cache—these are the loads that are “vulnerable” to external stores—update the signature with their address. External stores probe the signature. On a hit, they trigger a squash to the checkpoint. When a rally completes, the signature is cleared and the process repeats. Unlike signatures used to disambiguate speculative threads [4], enforce coarse-grain sequential consistency [5], or streamline conflict detection for transactions [22], iCFP signatures are not communicated between processors.

3.4. Other Implementation Issues

Simple runahead mode. As alluded to in Section 3.2, iCFP supports a “simple runahead” mode in which it uses the scratch register file to implement advance execution without miss-independent result commit. iCFP transitions to this mode whenever it runs out of slice buffer or store buffer entries or encounters a store with a poisoned (*i.e.*, unknown) address. When the corresponding condition resolves, iCFP resumes “full” advance execution.

Slice buffer management. iCFP requires that instructions in the slice buffer appear in program order. When combined with multi-threaded advance/rally, this implies that rally execution cannot dequeue instructions from the head of the slice buffer and then re-inject them at the tail as this would allow re-circulated slice instructions to interleave with new sliced instructions from the tail. Instead, iCFP marks a processed slice instruction as “un-poisoned”, and simply “re-poisons” its existing entry if the instruction has to be re-circulated. Rallying simply skips un-poisoned slice buffer entries. Banking the slice buffer with some degree that is higher than re-injection bandwidth reduces the bandwidth cost of skipping un-poisoned entries [13]. In iCFP, the slice buffer isn’t incrementally compacted, rather successive rally passes make it increasingly “sparse”, although entries can be reclaimed incrementally from the head. This makes the slice buffer somewhat more space inefficient than it otherwise could be, but it does enable several bandwidth optimizations including multi-threaded rally.

Exploiting additional poison bits. iCFP uses poison bits to track load misses and their dependent-instructions. When a miss returns, instructions that depend on the miss are processed. But so are younger instructions that depend on *any* miss, whether or not that miss has returned. iCFP can reduce this inefficiency by replacing poison bits with poison *bitvectors* where they

occur—in the register files, in the store buffer, and in the slice buffer.

Whenever a load miss is initially poisoned it is allocated a bit in a bitvector. Load misses to the same MSHR (*i.e.*, cache line) are allocated the same bit, whereas loads to different MSHRs *may* share a bit. The precise assignment of poison bits to MSHRs is unimportant, a simple round-robin scheme is sufficient. A register or store is considered poisoned if *any* poison bit in its poison bitvector is set and instructions are sliced out accordingly. Rallies are initiated by miss returns and so the processor knows which bits are being “un-poisoned”. Instructions which do not have any of these particular bits set are skipped—regardless of whether they have any other poison bits set.

The maximum number of useful poison bits is the degree of MLP iCFP can uncover—beyond software and hardware prefetch. Experiments show that programs can benefit from up to about 8 poison bits. 8 poison bits provide a 1.5% average performance gain over a single bit. *mcfl* sees a 6% benefit.

4. Some Comparisons

SLTP. SLTP (Simple Latency Tolerant Processor) implements non-blocking advance with blocking rallies and commit of miss-independent advance instructions [17]. SLTP’s blocking rally limitation comes from its register file design and register dependence tracking scheme. Specifically, SLTP uses a single register file and two checkpoints rather than two register files and a single checkpoint. It also tracks only poison information, not last writer identity. As a result it does not support partial updates to the main register file during slice re-execution. The main register file is “reconciled” only when the entire contents of the slice buffer have successfully re-executed.

SLTP also has a different data memory system, one based on the SRL (Store Redo Log) scheme [10]. Advance stores write their results into the SRL, a simple FIFO. Miss-independent stores also speculatively write to the data cache from where they can forward to miss-independent loads. When a rally begins, the speculative cache writes are discarded and the SRL is drained to the cache. Slice re-execution and SRL draining are interleaved in program order. The SRL design supports only “best effort” poison bit propagation—dependence prediction is used to propagate poison from miss-dependent stores to loads that forward from them. This speculation is verified by searching a large set-associative load queue. When an STLP rally completes, the speculative cache blocks are made non-speculative. iCFP’s data memory system—address-hash chained store buffer and

load signature—is both simpler and provides higher performance. We compare the two memory systems experimentally in Section 5.2.

Multipass Pipelining. “Flea-flicker” Multipass pipelining [3] is a different extension to Runahead execution. Whereas iCFP commits the results of miss-independent advance instructions and skips them during rallies, Multipass saves miss-independent results in a buffer and uses them to accelerate rallies by breaking data-dependences and increasing ILP. Section 5.1 compares iCFP to Multipass.

Rock. We don’t have many details about Sun’s Rock [26], but we know it implements a non-blocking in-order pipeline. Rock uses checkpoints and a slice buffer to defer miss-dependent instructions. Its implementation is described as two threads that leapfrog each other, passing the designation of architectural thread back and forth. It is not clear whether Rock makes multiple passes over the slice buffer or whether the architectural thread stalls when it encounters dependent misses. There is no description of Rock’s data memory system.

CPR/CFP, D-KIP, and TCI. iCFP is inspired by (out-of-order) CFP. CFP uses the out-of-order scheduler to capture and isolate the forward slices of L2 misses, preventing those instructions from tying up registers [13] and issue queue entries [24]. CFP targets L2 misses exclusively because the out-of-order engine can tolerate L2 hits.

CFP was initially implemented on top of a register-efficient checkpoint-based substrate (CPR) [1], but the CFP principle is easily applied to ROB-based substrates [7]. One ROB-based CFP implementation, D-KIP (Decoupled KILO-Instruction Processor) [20], re-executes L2-miss slices on a scalar in-order pipeline. Another, MSP (Multi-Scan Processor) [18] leverages multiple in-order pipelines to make multiple passes over the slice buffer. iCFP implements this functionality in a single in-order pipeline and uses it to tolerate misses at all cache levels. Its use of a single in-order pipeline also allows it to use a simple chained store buffer as opposed to a distributed load store queue [19].

TCI (Transparent Control Independence) [2] is a CFP derivative that uses in-order rename-stage slicing to reduce the branch mis-prediction penalty in an out-of-order processor. iCFP borrows the poison bitvector optimization from TCI.

ReSlice. ReSlice uses slice re-execution to reduce the cost of thread live-in mis-speculation in a speculatively multi-threaded architecture [23]. Re-slice tracks slices originating in different thread live-in separately, but can handle some slice overlaps. It can deal with some mem-

ory data hazards by recording values read by slice loads and over-written by slice stores. iCFP interleaves all slices in a single slice buffer. iCFP can deal with all slice overlaps and its use of a chained store buffer isolates slice re-execution from memory data hazards.

5. Experimental Evaluation

We evaluate iCFP using cycle-level simulation on the SPEC2000 benchmarks. The benchmarks are compiled for the Alpha AXP ISA at optimization level -O4. Benchmarks run to completion with 2% periodic sampling. Each 1 million instruction sample is preceded by a 4 million instruction cache and predictor warmup period. Our simulator cannot execute *fma3d* and *sixtrack*.

The timing simulator is based on the SimpleScalar 3.0 machine definition and system call modules. It simulates advanced branch prediction, an event-driven non-blocking cache hierarchy with realistic buses and miss-status holding registers (MSHRs), as well as hardware stream buffer prefetching. Table 1 describes our configuration in detail.

5.1. Comparative Performance

Figure 5 shows percent speedup over an in-order pipeline for Runahead, Multipass, SLTP, and iCFP. The different micro-architectures are configured similarly, but differ in the types of misses under which they block or advance. Runahead and SLTP advance under all L2 misses, but block on all—*i.e.*, both primary and secondary—data cache misses. Multipass advances under all L2 misses and primary data cache misses, but blocks on secondary data cache misses. iCFP advances under all primary and secondary misses. These settings produce the best results for each micro-architecture under our default configuration, specifically a 20-cycle L2 hit latency. Runahead and SLTP don’t advance under primary data cache misses because in both, there is a small cost relative to the baseline in-order pipeline—which stalls on the first miss-dependent instruction, not the miss itself—for advancing under a miss that doesn’t expose additional misses. In Runahead, this cost is incurred because the transition to advance mode happens immediately, causing instructions younger than the miss to be effectively discarded. In SLTP, the cost is associated with draining the SRL. With a 20-cycle L2 hit latency and a 10 stage pipeline, advance execution effectively has only 10 cycles to uncover an independent miss under a data cache miss. The chances of doing so are too low to overcome the small “startup” penalties associated with Runahead and SLTP.

On average (geometric mean over all of SPEC2000), iCFP improves performance by 16%, Multipass by 11%,

Bpred	24 Kbyte 3-table PPM direction predictor [14]. 2K-entry target buffer. 32-entry RAS.
Pipeline	10 stages: 3 I\$, 1 decode, 1 reg-read, 1 ALU, 3 D\$, 1 reg-write. 2-cycle fp-add, 4-cycle int/fp multiply
Execution	2-way superscalar, 2 integer, 1 fp/load/store/branch
I\$/D\$	32 Kbyte, 4-way set-associative, 64 byte line, with 8-entry victim buffer, 32-entry associative store buffer
L2	1 Mbyte, 8-way set-associative, 128-byte line, with 4-entry victim buffer, 20-cycle L2
Prefetchers	8 stream buffers with 8 128-byte blocks each
Memory	400 cycle latency to the first 16 bytes, 4 cycles to each additional 16 byte chunk. 64 outstanding misses
Runahead	256-entry runahead cache
Multipass	256-entry runahead cache, 128-entry instruction buffer, no compiler RESTART directives
SLTP	128-entry SRL, 128-entry slice buffer, idealized memory dependence prediction and load queue
iCFP	128-entry chained store buffer, 512-entry chain table, 128-entry slice buffer, 8-bit poison vectors

Table 1. Simulated processor configurations

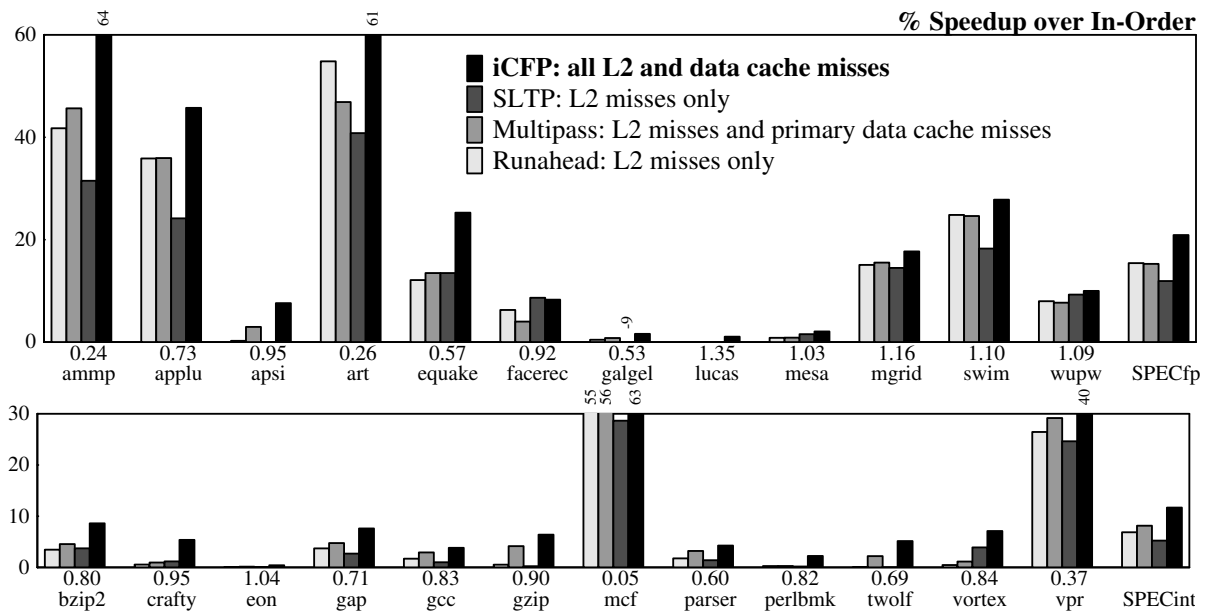


Figure 5. Runahead, Multipass, SLTP, and iCFP speedup over in-order

Runahead by 11%, and SLTP by 9%. For SPECfp, the four provide improvements of 21%, 15%, 15%, and 12%, respectively. SPECint improvements are 12%, 7%, 7%, and 5%. It is important to remember that the baseline processor includes stream buffer prefetching. It is also important to remember that the geometric means include several programs, *mesa*, *eon* and *vortex* to name three, which have few cache misses and essentially never enter advance mode—Table 2 contains a benchmark characterization which includes data cache and L2 misses per 1000 instructions. Programs with many data cache and L2 misses like *ammp*, *applu*, *art*, *mcf*, and *vpr* all see speedups of 40% or greater.

The important observation from Figure 5 is that iCFP matches or outperforms the three other schemes in every case except one minor exception—on *facerec*, SLTP outperforms iCFP by 0.3%. The effect in play here

is that, in SLTP, speculatively-written lines cannot be evicted. This perturbs the replacement sequence in a way that happens to reduce misses. Overall, however, SLTP’s use of an SRL-based data memory system severely limits its performance, and occasionally yields slow-downs over the baseline in-order pipeline (*e.g.* 9% on *galgel*). An SRL-based design requires speculatively written lines to be flushed from the data cache before a rally, potentially increasing D\$ misses and load latency (on *galgel*, load latency increases by 7%). An SRL-based scheme also requires that slice re-execution be inter-leaved with SRL draining, and this counter-acts most of the benefit of skipping miss-independent instructions. The SRL must also be completely drained before “tail” execution can resume. Finally, the requirement of single-pass blocking rallies constrains SLTP on programs with dependent misses, *e.g.*, *mcf* and *vpr*.

Bench	Miss/KI		D\$ MLP			L2 MLP			Rally/KI
	D\$	L2	iO	RA	iCFP	iO	RA	iCFP	
ammp	23	5	1.1	2.3	2.5	1	1.9	2	428
applu	21	3	2.2	5.4	5.9	2	5.5	5.9	105
apsi	19	0	2.2	2.2	2.7	6.8	4.2	4.2	49
art	122	19	2.6	23.9	43.6	1.8	18.4	35.5	951
equake	26	1	1.4	1.9	2.1	1.5	2.3	2.4	290
facerec	10	3	11	22.5	22.5	41.7	73.4	73.1	64
galgel	14	0	1.2	1.3	1.4	1.9	3.6	4.0	48
lucas	19	0	1.3	1.3	1.4	1.0	1.0	1.0	65
mesa	1	0	1.1	1.1	1.1	4.2	2.8	2.8	3
mgrid	13	0	1.5	3.4	4.7	1.5	7.8	12.0	15
swim	28	5	5	8.9	11.0	4.1	8.4	12.1	64
wupw	5	1	1.9	3.1	3.3	1.6	2.6	2.9	33
bzip2	5	1	2.0	2.5	2.5	3.2	3.6	3.8	32
crafty	4	0	1.0	1.1	1.1	1.0	1.2	1.2	29
eon	10	0	1.0	1.1	1.1	1.1	1.8	1.7	3
gap	5	1	1.5	1.9	2.0	2.8	2.4	2.4	29
gcc	11	0	1.3	1.6	1.6	3.6	2.9	2.9	38
gzip	11	0	1.1	1.2	1.5	8.3	8.0	8.8	94
mcf	115	46	3.1	4.6	5.0	2.9	4.2	4.5	2876
parser	10	1	1.0	1.1	1.1	1.1	1.1	1.1	238
perl	4	0	1.1	1.2	1.2	1.2	1.5	1.5	26
twolf	20	0	1.1	1.1	1.2	1.1	1.3	1.3	224
vortex	2	0	1.1	1.3	1.4	1.3	1.4	1.4	15
vpr	19	3	1.1	1.7	1.8	1.1	1.7	1.8	187

Table 2. iCFP diagnostics

Rally overhead. iCFP outperforms Runahead and Multipass because its rallies can skip miss-independent advance instructions. Multipass has a limited form of rally acceleration (dependence-breaking) and usually slightly out-performs Runahead. Table 2 shows the number of instructions iCFP re-executes in rally mode per 1000 program instructions. This number can be greater than 1000 because iCFP makes multiple rally passes—for long chains of dependent misses (e.g., *mcf*) iCFP makes as many rally passes as there are dependent misses in the chain. Nevertheless, iCFP’s rally overhead is lower than that of Runahead and Multipass.

MLP. Faster rallies and uniform non-blocking also help increase MLP. Table 2 shows data cache and L2 MLP for in-order (iO), Runahead (RA), and iCFP. iCFP boosts MLP over both in-order and Runahead in almost all cases. Note, our simulated processor can only practically exploit an L2 MLP of 12, because of the ratio of memory latency (400 cycles) to memory bus bandwidth (one L2 cache line every 32 cycles).

Tolerating all-level cache misses. The main claim of this paper (see title) is that iCFP’s combination of features—minimal rallies and uniform non-blocking—allow it to tolerate both short and long cache misses in an in-order processor. Our results support this claim. The examples in Figure 1 should provide some intuition. To gain further insight, we repeat our Runahead and iCFP experiments with different L2 hit latencies. Figure 6

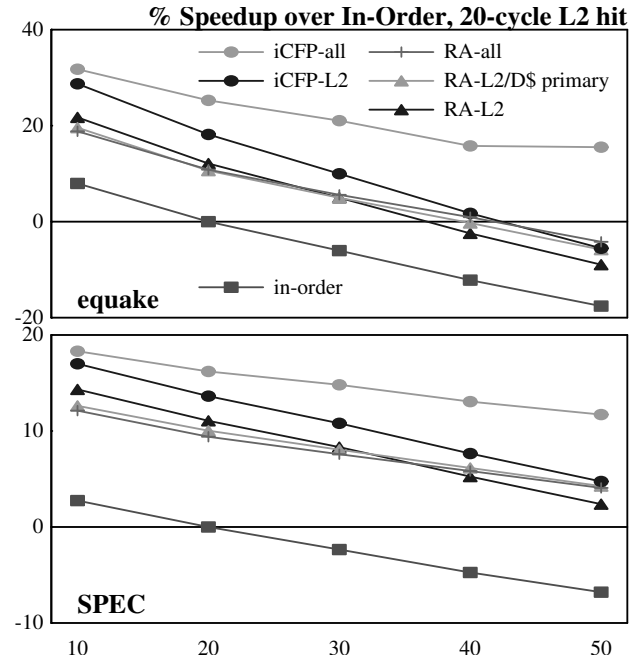


Figure 6. L2 hit-latency sensitivity

shows results for benchmark *equake* and for geometric mean over all of Spec. We experiment with two different iCFP configurations, one that advances only on L2 misses and one that advances under all misses. We also show three Runahead configurations, one that advances under L2 misses only, one that also advances under primary data cache misses, and one that advances under all misses, including secondary data cache misses.

The SPEC average results justify our choice of L2-only advance as the Runahead configuration. They also confirm the intuition that at higher L2 hit latencies, allowing Runahead to advance on data cache misses becomes profitable. The *equake* results illustrate the secondary data cache miss dilemma faced by Runahead. At short L2 hit latencies, *equake* prefers that Runahead block on secondary data cache misses. At higher L2 hit latencies, it prefers that Runahead advance on those misses. In iCFP, advancing on any data miss is profitable at virtually any L2 hit latency.

5.2. Feature Contribution Analysis

iCFP feature “build”. We performed additional experiments to isolate the performance contributions of iCFP’s various features. Figure 7 shows these experiments as a “build” from SLTP, the leftmost bar in the graph. All bars in this build allow advance execution on any miss, as iCFP does. The second bar replaces SLTP’s SRL-based memory system with iCFP’s chained store buffer. In itself, the chained store buffer provides

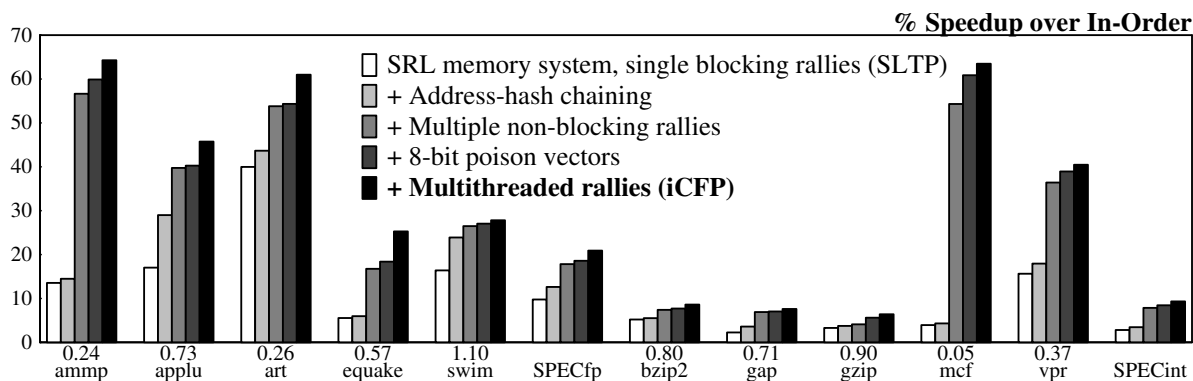


Figure 7. iCFP feature performance analysis

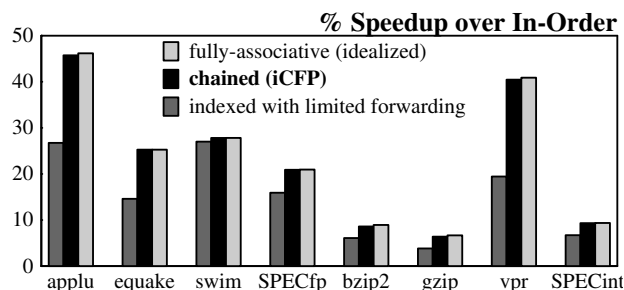


Figure 8. Store buffer effects

an average performance gain of about 2%, although individual programs (e.g., *applu* and *swim*) benefit more significantly. The bigger contribution of the chained store buffer is that it enables non-blocking rallies, which are added in the third bar. Non-blocking rallies add an average of 7% to performance and greatly improve the performance of programs with many dependent misses (e.g., *mcf* and *vpr*). Using 8-bit poison vectors instead of singleton poison bits allows rallies to skip instructions that are independent of the particular miss that just returned. The final feature—although it requires no additional support over non-blocking rallies—is the ability to multi-thread rallies with execution of new instructions at the tail.

Store buffer alternatives. Figure 8 compares our chained store buffer to two other designs: an idealized, fully-associative store buffer, and an indexed store buffer that supports limited forwarding. In the limited forwarding scheme, the pipeline stalls if a load “hits” in the chain table but doesn’t match the address of the corresponding store—this is the iCFP equivalent of out-of-order CFP’s SRL/LCF scheme [10]. Intuitively, this configuration performs poorly. In out-of-order CFP, younger instructions can flow around a stalled load; in iCFP, they cannot. More surprising is that chaining closely tracks the performance of idealized fully-

associative search. The difference between these two is less than 1% for every program. The reason is that the average number of “excess” store buffer hops per load is low. The only two benchmarks which average more than 5 extra hops per hundred committed loads are *ammp* (18) and *art* (47). Chaining performance is a function of chain table size. A 64-entry chain table reduces performance—relative to a 512-entry table—by 0.3% on average with a maximum of 4% (*ammp*).

5.3. Area Overheads

We use a modified version of CACTI-4.1 [25] to estimate the area overheads of Runahead, Multipass, SLTP, and iCFP in 45nm technology. Runahead overhead includes the poison bits and Runahead cache. Multipass overhead includes poison bits, result buffer, forwarding cache, and load disambiguation unit. SLTP overhead includes poison bits, SRL, and load queue (we do not count the memory dependence predictor). iCFP overhead comprises poison bits, sequence numbers, store buffer, chain table, and signature. We do not count the scratch register file as overhead because it also supports multi-threading. We do count the cost of the shadow-bitcell checkpoints which we estimate for a 6-port register file using the proposed layout [9].

Assuming 128-entry slice/result buffers, a 512-entry chain table, 8-bit poison vectors, 10-bit sequence numbers, a 256-entry forwarding cache, and a 256-entry load queue, we estimate the area overheads of Runahead, Multipass, SLTP, and iCFP as 0.12, 0.22, 0.36, and 0.26 mm², respectively. These footprints are small relative to the area of a 2-way issue in-order processor (with floating-point) which we estimate to be between 4 and 8 mm² in this technology. Certainly, iCFP’s performance advantages over Runahead and Multipass justify its marginal area cost. iCFP out-performs SLTP despite a smaller area footprint.

Additional experiments show that a 2-way issue out-

of-order processor has a 68% performance advantage over our 2-way in-order pipeline, while a 2-way issue (out-of-order) CFP pipeline has an 83% advantage. Certainly these are greater than the 16% advantage that iCFP provides. However, these designs need somewhat more than 0.26 mm² to provide their respective gains.

6. Conclusions

Due to power concerns, multi-threaded in-order processors are beginning to replace out-of-order processors even in high-performance chips. In this paper, we show how to use an additional thread context to recoup some of the single-thread performance lost in this transition. We describe iCFP, an in-order implementation of continual flow pipelining which uses an additional register file to execute deferred miss-dependent instructions.

iCFP is related to previous proposals like Runahead execution, Multipass pipelining, and SLTP (Simple Latency Tolerant Processor). But it contains a unique combination of features not found in any single previous proposal. First, it supports non-blocking under all types of cache misses, primary, secondary, and dependent. Second, when advancing under any miss, it can “commit” all miss-independent instructions and skip them during subsequent passes of the same code region. This feature combination is enabled by an enhanced register dependence tracking mechanism and a novel store buffer design, and it allows iCFP to effectively tolerate misses at any cache level.

7. Acknowledgments

We thank the reviewers for their comments on this submission. This work was supported by NSF grant CCF-0541292 and by a grant from the Intel Research Council. Andrew Hilton was partially supported by a fellowship from the University of Pennsylvania Center for Teaching and Learning.

References

- [1] H. Akkary, R. Rajwar, and S. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proc. 36th Intl. Symp. on Microarchitecture*, pages 423–434, Dec. 2003.
- [2] A. Al-Zawawi, V. Reddy, E. Rotenberg, and H. Akkary. Transparent Control Independence (TCI). In *Proc. 34th Intl. Symp. on Computer Architecture*, pages 448–459, Jun. 2007.
- [3] R. Barnes, S. Ryoo, and W.-M. Hwu. “Flea-Flicker” Multipass Pipelining: An Alternative to the High-Powered Out-of-Order Offense. In *Proc. 38th Intl. Symp. on Microarchitecture*, pages 319–330, Nov. 2005.
- [4] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proc. 33rd Intl. Symp. on Computer Architecture*, pages 227–238, Jun. 2006.
- [5] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *Proc. 34th Intl. Symp. on Computer Architecture*, pages 278–289, Jun. 2007.
- [6] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-Performance Throughput Computing. *IEEE Micro*, 25(3):32–45, May 2005.
- [7] A. Cristal, O. Santana, M. Valero, and J. Martinez. Toward KILO-Instruction Processors. *ACM Trans. on Architecture and Code Optimization*, 1(4):389–417, Dec. 2004.
- [8] J. Dundas and T. Mudge. Improving Data Cache Performance by Pre-Executing Instructions Under a Cache Miss. In *Proc. 1997 Intl. Conf. on Supercomputing*, pages 68–75, Jun. 1997.
- [9] O. Ergin, D. Balkan, D. Ponomarev, and K. Ghose. Increasing Processor Performance Through Early Register Release. In *Proc. 22nd IEEE Intl. Conf. on Computer Design*, pages 480–487, Oct. 2004.
- [10] A. Gandhi, H. Akkary, R. Rajwar, S. Srinivasan, and K. Lai. Scalable Load and Store Processing in Latency Tolerant Processors. In *Proc. 32nd Intl. Symp. on Computer Architecture*, pages 446–457, Jun. 2005.
- [11] K. Krewell. Sun’s Niagara Pours on the Cores. *Microprocessor Report*, 18(10):11–13, Sept. 2004.
- [12] H. Le, W. Starke, J. Fields, F. O’Connell, D. Nguyen, B. Ronchetti, W. Sauer, and E. S. M. Vaden. POWER6 Microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, Nov. 2007.
- [13] A. Lebeck, J. Koppaialil, T. Li, J. Patwardhan, and E. Rotenberg. A Large, Fast Instruction Window for Tolerating Cache Misses. In *Proc. 29th Intl. Symp. on Computer Architecture*, pages 59–70, May 2002.
- [14] P. Michaud. A PPM-like, Tag-Based Branch Predictor. *Journal of Instruction Level Parallelism*, 7(1):1–10, Apr. 2005.
- [15] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: Log-Based Transactional Memory. In *Proc. 12th Intl. Symp. on High-Performance Computer Architecture*, Jan. 2006.
- [16] O. Mutlu, J. Stark, C. Wilkerson, and Y. Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors. In *Proc. 9th Intl. Symp. on High Performance Computer Architecture*, pages 129–140, Feb. 2003.
- [17] S. Nekkalapu, H. Akkary, K. Jothi, R. Retnamma, and X. Song. A Simple Latency Tolerant Processor. In *Proc. 26th Intl. Conf. on Computer Design*, Oct. 2008.
- [18] M. Pericas, A. Cristal, F. Cazorla, R. Gonzalez, D. Jimenez, and M. Valero. A Flexible Heterogeneous Multi-Core Architecture. In *Proc. 12th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sep. 2007.
- [19] M. Pericas, A. Cristal, F. Cazorla, R. Gonzalez, A. Veidenbaum, D. Jimenez, and M. Valero. A Two-Level Load/Store Queue Based on Execution Locality. In *Proc. 12th Intl. Symp. on Computer Architecture (to appear)*, Jun. 2008.
- [20] M. Pericas, R. Gonzalez, D. Jimenez, and M. Valero. A Decoupled KILO-Instruction Processor. In *Proc. 12th Intl. Symp. on High Performance Computer Architecture*, pages 53–64, Feb. 2006.
- [21] A. Roth. Store Vulnerability Window (SVW): Re-Execution Filtering for Enhanced Load Optimization. In *Proc. 32nd Intl. Symp. on Computer Architecture*, pages 458–468, Jun. 2005.
- [22] D. Sanchez, L. Yen, M. Hill, and K. Sankaralingam. Implementing Signatures for Transactional Memory. In *Proc. 40th Intl. Symp. on Microarchitecture*, Nov. 2007.
- [23] S. Sarangi, W. Liu, J. Torrellas, and Y. Zhou. Re-Slice: Selective Re-Execution of Long-Retired Misspeculated Instructions using Forward Slicing. In *Proc. 38th International Symp. on Microarchitecture*, pages 257–268, Dec. 2005.
- [24] S. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton. Continual Flow Pipelines. In *Proc. 11th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 107–119, Oct. 2004.
- [25] D. Tarjan, S. Thoziyoor, and N. Jouppi. CACTI 4.0. Technical Report HPL-2006-86, Hewlett-Packard Labs Technical Report, Jun. 2006.
- [26] M. Tremblay and S. Chaudhry. A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout Thread CMT SPARC Processor. In *Proc. 2008 IEEE International Solid-State Circuits Conf.*, Feb. 2008.