Slipstream Memory Hierarchies

Technical Report CESR-TR-02-3

Zach Purser, Karthik Sundaramoorthy, Eric Rotenberg
Center for Embedded Systems Research (CESR)
Department of Electrical and Computer Engineering
North Carolina State University
{zrpurser, ksundar, ericro}@ece.ncsu.edu

Abstract

A slipstream processor harnesses an otherwise unused processing element in a chip multiprocessor (CMP) to speed up a single program. It does this by running two redundant copies of the program. Predicted-non-essential computation is speculatively removed from one of the programs, speeding it up. The second program checks the forward progress of the first and is also sped up in the process. Both program copies finish sooner than either can alone.

The redundant programs are architecturally independent and this leads to a simple execution model. Physical memory pages are duplicated by the operating system, sparing the processor from explicitly managing a fixed amount of transparent rename storage. Unfortunately, doubling memory usage partially negates performance gains. We observe that 1) the already-replicated L1 caches in a CMP provide enough implicit rename storage and 2) this storage does not need to be explicitly managed because the slipstream paradigm is tolerant of slightly inaccurate memory renaming. Leveraging unmodified cache actions within a typical private-L1/shared-L2 memory hierarchy, we develop an efficient hardware-based memory duplication approach that significantly outperforms software-based duplication, yet does not require any explicit hardware management.

Furthermore, the new duplication approach enables much simpler state recovery when the speculative program diverges. Simple cache flushing eliminates a previously-required slipstream recovery component. And the performance impact of flush-induced compulsory misses is reduced by exploiting preserved data within flushed cache lines as highly-accurate value predictions.

1. Introduction

Chip multiprocessing (CMP) [12,17,29] and simultaneous multithreading (SMT) [8,30,31] are compelling because they maximize the performance capacity of a single chip by evolutionary rather than revolutionary means. Independent jobs or parallel tasks that otherwise execute on physically separate processors now execute on the same chip. Several companies have already announced CMP [e.g.,12,29] or SMT [e.g.,8,27] designs because of their high payoff and conceptual simplicity. Unfortunately, CMP/SMT processors do not explicitly address the performance of individual sequential programs. Yet, historically, improving single-program performance is crucial for remaining competitive.

The slipstream paradigm [19,21,22,28] harnesses a CMP/SMT processor to improve single-program performance without fundamentally changing its operation. A slipstream processor concurrently runs two redundant copies of the program. One of the programs always runs slightly ahead of the other. The leading program is called the *advanced stream*, or A-stream, and the trailing program is called the *redundant stream*, or R-stream. Hardware monitors the R-stream and detects dynamic instructions that, in retrospect, were unnecessary for correct forward progress. This knowledge is later used to speculatively but accurately reduce the A-stream: predicted-non-essential instructions are bypassed in the A-stream if there is high confidence correct forward progress can still be made. Infrequent deviations are detected by passing all A-stream control and data flow outcomes to the non-speculative R-stream, which checks the outcomes against its own. When the R-stream detects a difference, corrupted architectural state of the A-stream is restored to match the R-stream.

Prior work demonstrated the A-stream/R-stream arrangement often leads to an overall performance improvement [19]. The A-stream is sped up because it fetches and executes fewer instructions. The R-stream is not reduced in terms of retired instructions. However, it fetches and executes more efficiently because outcomes from the A-stream are used as ideal predictions. The A-stream and R-stream finish at virtually the same time — the R-stream follows closely behind the A-stream — and both finish sooner than a single copy of the program would.

The appeal of slipstreaming is that the underlying CMP/SMT processor operates on the A-stream and R-stream as if they were unrelated. Conventional register and memory dependence mechanisms remain intact because the programs are architecturally independent. Unfortunately, independence comes at the expense of *doubling memory usage*. That is, the easiest way to ensure A-stream loads/stores do not interfere with R-stream loads/stores is to have the operating system allocate separate physical memory pages for each program. Then, the processor does not have to rename memory locations itself. Hardware memory renaming is typically complex because the processor must provide the illusion of unlimited memory to one of the program copies, within a fixed amount of hardware-managed renaming storage.

Therein lies the challenge in the design of memory systems for slipstream processors. On the one hand, software-based memory duplication leads to a simple execution model. On the other hand, full duplication may degrade memory system performance.

In this paper, we make two major contributions.

1. *We develop a hardware-based solution that duplicates memory efficiently, yet retains the simplicity of software-based memory duplication.*

   We exploit a memory hierarchy similar to the IBM Power4 [12], a commercial CMP composed of two processing elements. Our hierarchy has a private level-1 (L1) cache per processing element and the L1 caches write-through to a shared level-2 (L2) cache. Our approach works as follows. First, the private L1 caches implicitly provide unique storage for the A-stream and R-stream, therefore, extra storage for renaming memory locations is not explicitly provided. Second, we make one simple change to the policy of the A-stream's L1 cache: it is neither write-through nor write-back, i.e., the A-stream never writes to the L2 cache. If the A-stream writes to a line in its L1 cache, and later that line is replaced, *the update is simply lost*. Dropping A-stream L1 cache updates works due to the nature of slipstreaming. By the time the A-stream re-references the evicted line, the R-stream is likely to have performed the corresponding redundant store to its L1 cache and the L2 cache (R-stream L1 is write-through). When the A-stream re-references the line from the L2 cache, the line most likely reflects the previously-lost A-stream update. Moreover, even if the R-stream has not performed the corresponding redundant store before the A-stream re-references the line, the A-stream is speculative in any case and this just adds another source of A-stream misspeculation.

   From the perspective of hardware, the above approach is virtually identical to software-based memory duplication. As before, no special hardware mechanisms are required: 1) rename storage is not explicitly provided due to already-replicated L1 caches [10,13,18,26], and 2) the fixed A-stream storage is not explicitly managed because *renaming does not have to be 100% accurate*. The latter point is a key departure from classical renaming approaches. Classically, hardware proactively determines when precious storage can be freed. In the context of slipstream processors, an A-stream L1 line can be "freed" when the corresponding R-stream L1/L2 line becomes redundant with it. But we do not have to precisely identify when this happens because we can afford to be incorrect (a general mechanism is in place to detect A-stream deviations, regardless of the cause). Instead, we passively free A-stream storage in the normal course of line replacement, essentially making a *prediction* at replacement time that either the L2 line is redundant with the A-stream L1 line or will be before re-referencing it.

2. *We simplify restoring memory locations when the A-stream diverges, including eliminating a previously required slipstream processor component (memory recovery controller).*

   Occasionally, the A-stream does not make correct forward progress and its state becomes corrupted. A recovery sequence restores A-stream state so it matches the R-stream. Register state is finite and restoring all registers is feasible. We do not have this luxury with memory: special hardware pin-points memory locations that need to be restored [28]. With software-based memory duplication, sophisticated recovery is unavoidable because A-stream and R-stream physical memory pages are distinct. With our new duplication approach, however, A-stream memory can be restored simply by flushing the A-stream L1 cache. Then, A-stream and R-stream memory match exactly since the A-stream must re-reference all data in the L2 cache, which is R-stream-only data. We also develop a novel technique for reducing the performance impact of flush-induced compulsory misses in the A-stream. A line is flushed by marking it invalid, but both the tag and data are preserved. Preserved values are usually correct and may be consumed as accurate value predictions, allowing load-miss-dependent instructions to execute while the flushed lines are re-filled from the L2 cache.

The original slipstream microarchitecture (using software-based memory duplication) is reviewed in Section 2. The two alternative memory duplication models are described in Section 3. We also describe three recovery models in Section 3 — the original recovery controller and two simpler flush models enabled by hardware-based duplication. Section 4 describes the simulator and benchmarks, and simulation results comparing duplication and recovery methods are presented in Section 5. Related work is discussed in Section 6.

2. Review of slipstream microarchitecture

We review the slipstream microarchitecture in this section. Detailed descriptions are omitted to conserve space, but enough background is provided to reason about alternative memory hierarchies, the focus of this paper. For reference, more detailed microarchitecture descriptions are available elsewhere [19].

A slipstream processor implemented on a 2-way CMP is shown in Figure 1. Existing superscalar cores are shown within shaded boxes. Four new slipstream components are shown external to and interfacing with the superscalar cores: instruction-removal detector (IR-detector), instruction-removal predictor (IR-predictor), delay buffer, and recovery controller. The superscalar core is designed to run either the A-stream or the R-stream (we arbitrarily show the A-stream on the left and R-stream on the right). Symmetric interfaces to/from the fetch unit and execution pipeline of both cores makes designing a single core natural.

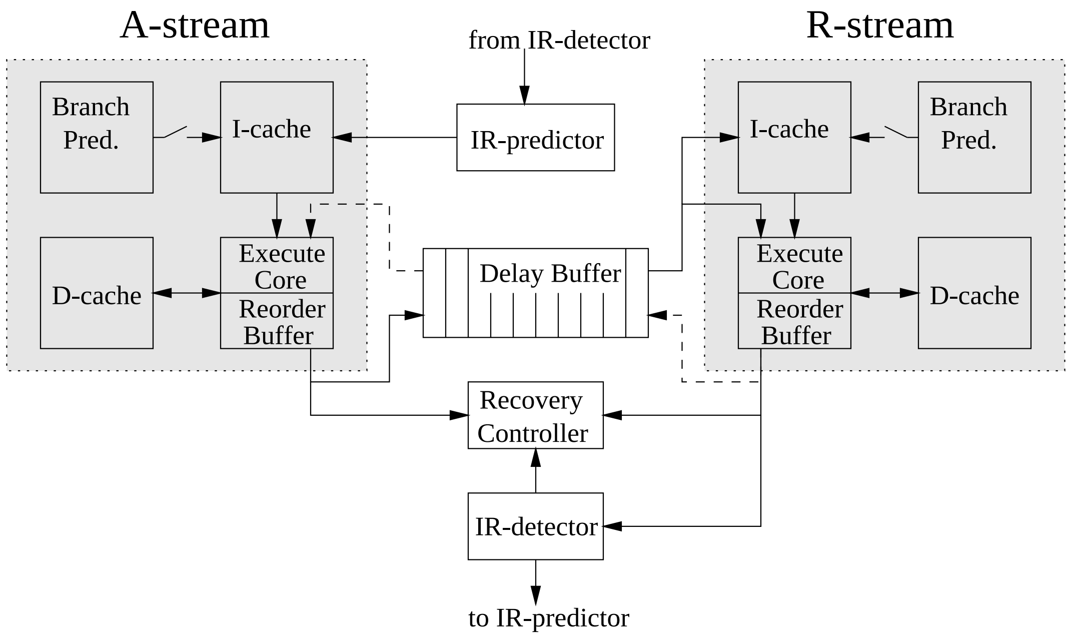

FIGURE 1. Slipstream microarchitecture.

2.1 Reducing the A-stream: IR-detector and IR-predictor

The A-stream is reduced by a process called instruction-removal, coordinated by the IR-detector and IR-predictor.

The IR-detector monitors retired R-stream instructions. It detects instructions that could have been safely removed and, therefore, might possibly be removed in the future. Candidates for removal include 1) ineffectual writes (register/memory writes that are not referenced before being overwritten, or writes that do not change the value in a location) and 2) correctly-predicted branches. The IR-detector contains a small shifting window of dynamic instructions arranged in program order. Circuitry among the instructions is configured to form connections between producer and consumer instructions, forming a reverse data flow graph (R-DFG). As newly-retired R-stream instructions are merged into the R-DFG, the IR-detector selects ineffectual writes and correctly-predicted branches for removal. Then, the R-DFG circuitry automatically back-propagates this removal status to producer instructions. A producer instruction is also selected for removal if all of its consumers are selected.

The IR-predictor is a modified branch predictor. Like a conventional branch predictor, it produces the program counter (PC) of the next fetch block. However, the predicted next PC reflects skipping any number of dynamic instructions that a conventional processor would otherwise fetch. The IR-predictor also indicates which instructions within a fetch block can be removed after fetch and before decode. The IR-detector indicates to the IR-predictor which dynamic instructions might be removed in the future. Repeated indications by the IR-detector build up confidence in the IR-predictor, and future instances of the ineffectual writes, predictable branches, and computation chains leading up to them are removed from the A-stream.

2.2 Delay buffer

The delay buffer is a FIFO queue that passes retired A-stream outcomes to the R-stream for checking. Although branch-predictable computation is removed from the A-stream, the implied branch predictions are still produced by the IR-predictor. Thus, the delay buffer contains a complete history of control flow, only a subset of which was verified by A-stream computation. The delay buffer contains a partial history of data flow since the A-stream executes only a subset of the dynamic instruction stream. Branch predictions from the delay buffer drive the R-stream fetch unit. Value predictions from the delay buffer are injected into the R-stream pipeline as instructions are renamed/dispatched. The delay buffer also contains 1-bit per original instruction indicating whether or not the instruction was skipped by the A-stream; removal information is needed to route value predictions of non-skipped instructions to the correct R-stream instructions.

2.3 IR-mispredictions and recovery

An instruction-removal misprediction, or IR-misprediction, occurs when instructions were removed from the A-stream that should not have been. IR-mispredictions cause the A-stream to no longer make correct forward progress. IR-mispredictions are detected as branch/value mispredictions in the R-stream. By that time, the A-stream has corrupted its architectural register and memory state. Recovery involves re-synchronizing the A-stream with respect to the R-stream. All A-stream registers are restored by copying values from the R-stream registers. Restoring memory is more complicated because corrupt locations must be pin-pointed. At all times, the *recovery controller* monitors store activity in the A-stream and R-stream. It uses this information to maintain a minimal, up-to-date list of A-stream memory locations that may need to be restored from the R-stream in the event of an IR-misprediction. A-stream registers/memory locations are restored by copying values from the corresponding R-stream registers/memory locations. The delay buffer, used in the reverse direction, provides a convenient datapath for copying restoration values from R-stream to A-stream.

3. Slipstream memory hierarchies

Slipstreaming requires memory duplication so that A-stream loads/stores and R-stream loads/stores do not interfere with each other. In Section 3.1, we examine both software-based duplication and our new hardware-based duplication approach. Section 3.2 describes three memory recovery models, including the recovery controller and two much simpler cache-flush approaches enabled by hardware-based duplication.

3.1 Memory duplication models

3.1.1 Software-based memory duplication

In previous slipstream implementations [19,28], the operating system duplicated physical memory pages to separate the two redundant programs. Software-based memory duplication is shown in Figure 2. Light shading and dark shading indicate data from A-stream pages and R-stream pages, respectively. As the figure indicates, a major drawback of software-based duplication is the capacity of the memory hierarchy is effectively halved (assuming physical memory is fully utilized to begin with). Some or all of the performance improvement due to slipstreaming may be negated due to additional capacity/conflict misses throughout the memory hierarchy. In large commercial systems (e.g., database servers), doubling physical memory requirements of all running programs may be unacceptable.

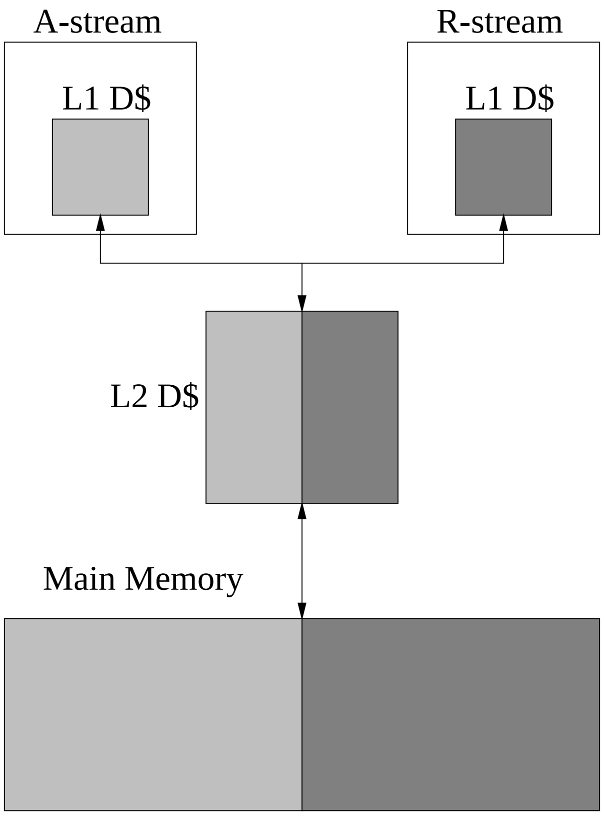

FIGURE 2. Software-based memory duplication.

Another major drawback of software-based memory duplication is that it exposes slipstream to the operating system (O/S). The O/S must duplicate the original program's physical memory image and manage separate page tables for the A-stream and R-stream. And trap handling requires unique O/S support. "Trap" refers to system calls, exceptions, and external interrupts. Slipstream processors synchronize the A-stream and R-stream at a trap [21]. After that, the O/S has two options. In the first option, the O/S preserves both programs, i.e., both programs are swapped out and later swapped in. The O/S handles the trap as it would for one program, but duplicates the results as needed for both suspended contexts. For example, file input/output (I/O) is not duplicated but the memory state of both contexts should reflect the results of file I/O. In the second option, one of the program copies is killed by the O/S before servicing the trap. The remaining program is swapped out and the trap is serviced in the usual manner. To restart, the suspended program is duplicated and both copies are swapped in. The first option complicates servicing of traps, whereas the second option incurs high performance overhead because all pages must be copied after every trap. The fact that the O/S is involved at all is undesirable.

3.1.2 Hardware-based memory duplication

The A-stream reaches store instructions before the R-stream, so R-stream memory state lags slightly behind A-stream memory state — *but not by much*. The redundant programs are typically no more than a few thousand instructions apart, and often less. This means the vast majority of A-stream and R-stream physical pages in Figure 2 are identical, and only a small amount of duplication is required. Below, we describe how a representative CMP memory hierarchy can be leveraged with almost no modifications to support efficient and system-transparent slipstreaming.

1. The L1 data cache is typically replicated in a CMP, a private cache for each processing element (PE), providing convenient and implicit memory replication close to the PEs. Both the A-stream and R-stream read/write their respective L1 caches independently and freely.

2. We assume the CMP has a single, shared L2 cache.

3. The R-stream L1 cache is write-through. That is, when the R-stream performs a store in the L1 cache, it also performs the store in the L2 cache.

4. The A-stream L1 cache is neither write-through nor write-back. When the A-stream performs a store in the L1 cache, it does *not* also perform the store in the L2 cache. Furthermore, if a "dirty" line (a line that was stored to) needs to be evicted to make room for another cache line, the line is *not* written back to the L2 cache. The evicted cache line, and the updated data it contains, is simply lost. In short, the A-stream can read from the L2 cache but not write to it.

Figure 3 shows hardware-based memory duplication. As before, A-stream state is indicated with light shading and R-stream state with dark shading. A thin black rectangle in the A-stream L1 cache represents a cache line to which the A-stream has performed a store, and the R-stream has not yet performed its corresponding redundant store. The figure shows how one such cache line written by the A-stream is evicted from the cache and the update it contains is lost, because the A-stream does not have its own renamed state beyond the L1 cache.

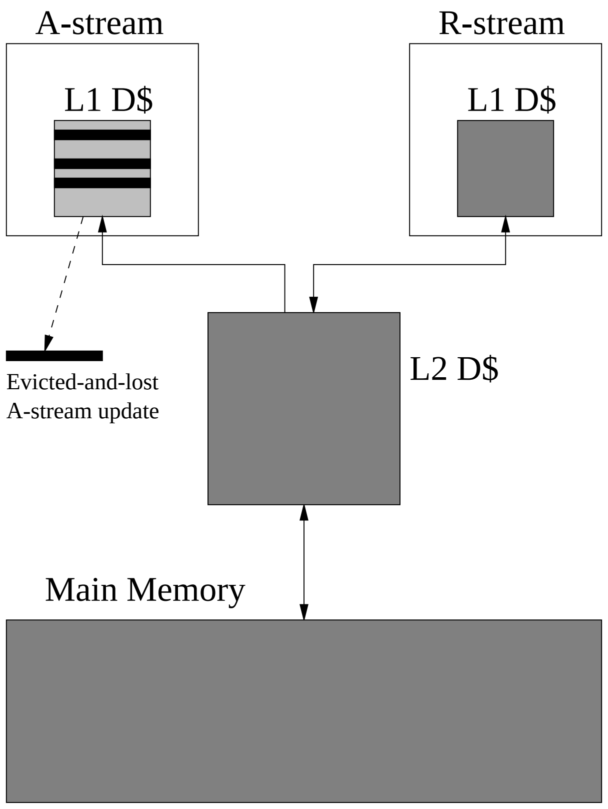

FIGURE 3. Hardware-based memory duplication.

Evicting and losing A-stream updates is rarely a problem because the R-stream usually reproduces the data before the A-stream re-references it. The R-stream is generally not far behind and, because its L1 cache is write-through, the evicted-and-lost A-stream data is re-created in the L2 cache before the A-stream needs it again. Occasionally, the A-stream re-references the line in the L2 cache before the R-stream has performed its corresponding update. The A-stream gets stale data, but the A-stream is speculative in any case and A-stream mispredictions are recoverable. In Section 5, we measure how many stale bytes are consumed by the A-stream; we label this measurement *stale* in Section 5.

Losing A-stream data can occasionally aid the A-stream. The A-stream sometimes incorrectly skips a store or incorrectly performs a store. In either case, the corrupt cache line may be evicted before being referenced and, later, the A-stream references a correct version of the cache line from the L2 cache. In Section 5, we measure how many inadvertently-repaired bytes are referenced by the A-stream; we label this measurement *self-repair* in Section 5.

In summary, our new approach is as simple as before because the processor does nothing explicit to manage A-stream storage. Instead, inherent cache actions perform the desired operations: the A-stream *implicitly* duplicates memory by allocating lines in its L1 cache and *implicitly* frees memory by replacing lines in its L1 cache. The approach is also efficient: memory hierarchy performance is not impacted because duplication is limited to the already-replicated L1 cache.

We should point out that a write-through R-stream L1 cache is not a strict requirement. If the R-stream L1 cache were write-back, coherence mechanisms would ensure the A-stream gets the most recent R-stream data when the A-stream fetches from the L2. Either the R-stream L1 cache snoops on its own behalf or, more typically, the L2 cache maintains inclusion and snoops on behalf of the L1 caches. We did not also implement write-back due to time constraints.

3.2 Recovery models

3.2.1 Recovery controller

The *recovery controller* [28] monitors store activity in the A-stream, R-stream, and IR-detector, pin-pointing memory locations that need to be restored from the R-stream if the A-stream goes astray. It maintains a list of memory locations, identified by address, that are known to differ between the A-stream and R-stream. Actually, there are many locations that differ but do not affect program correctness. The recovery controller only tracks memory locations that differ and have not yet been verified as “OK” to differ. Our implementation keeps track of individual doublewords, although a word or cache line granularity could also be used.

Figure 4 demonstrates how the recovery controller tracks memory locations. The figure shows the progression in time (from left to right) of two different types of stores as they pass first through the A-stream, then through the R-stream, and finally through the IR-detector. The contents of the recovery controller is shown evolving over time, at the bottom of the figure. The first store is to address *A* and is not skipped by the A-stream; it is shown with a solid circle to indicate it was not skipped. The second store is to address *B* and is skipped by the A-stream; it is shown with a hollow circle to indicate it was skipped.

When store(*A*) is committed by the A-stream, address *A* is added to the recovery controller because that location now differs between the A-stream and R-stream. When the R-stream commits the corresponding redundant store(*A*), address *A* is removed from the recovery controller because it no longer differs. If the processor initiates a recovery sequence after store(*A*) is committed by the A-stream and before it is committed by the R-stream, we know to “undo” the A-stream store because address *A* is in the recovery controller’s list.

Store(*B*) is skipped by the A-stream so no signal is sent by it to the recovery controller. The R-stream knows which instructions were skipped by the A-stream (as described in Section 2.2, the delay buffer contains removal information for matching up results of A-stream-executed instructions with R-stream instructions). When store(*B*) is committed by the R-stream, address *B* is added to the recovery controller because the A-stream perhaps should have performed the store. By the time store(*B*) has reached the end of the IR-detector’s analysis scope, we will have determined whether or not store(*B*) was necessary. If store(*B*) is deemed ineffectual, address *B* is removed from the recovery controller. Otherwise, it remains in the recovery controller until the next recovery sequence since we do not know for certain that it was alright to skip store(*B*).

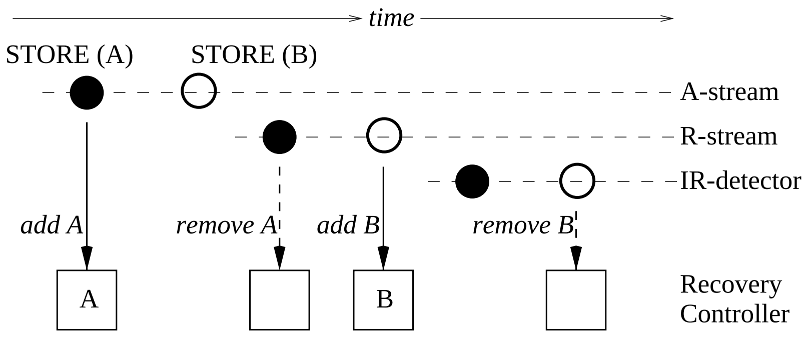

FIGURE 4. Recovery controller operation.

The recovery controller is organized as a set-associative or fully-associative buffer. Each entry contains a memory address and two counters, the *store-undo* counter and the *store-do* counter. Counters are used because there can be multiple unverified stores to the same location, all of which must be tracked. Referring back to Figure 4, the *store-undo* counter for address *A* is incremented when store(*A*) is committed by the A-stream and decremented when store(*A*) is committed by the R-stream. The *store-do* counter for address *B* is incremented when store(*B*) is committed by the R-stream and decremented when store(*B*) is detected as removable by the IR-detector. An entry in the recovery controller can be replaced when both the *store-undo* and *store-do* counts are zero. If there is no room for a new address in the recovery controller, a recovery sequence is initiated to clear out the recovery controller.

The recovery controller is required if software-based duplication is used. It also works with hardware-based duplication but is optional, because simpler recovery models are enabled by efficient duplication. Actually, the recovery controller does not recover state perfectly when used in conjunction with hardware-based duplication. Namely, *the recovery controller has no knowledge of stale lines brought into the A-stream L1 cache* (A-stream evicts and loses an updated line, and re-references a stale version of the line from the L2 cache). After a recovery, stale data may persist in the cache and potentially cause problems for the A-stream in the future. In Section 5, we measure how many stale bytes introduced before recovery are referenced by the A-stream after recovery; we label this measurement *persistent-stale* in Section 5.

3.2.2 Flush cache

The recovery controller adds complexity to slipstream processors. The new hardware-based memory duplication approach can be exploited to eliminate the recovery controller. A-stream memory state can be restored, i.e., re-synchronized with the memory state of the R-stream, simply by invalidating *en masse* the A-stream's L1 cache lines. Recovery is gradual as lines are re-accessed from the L2 cache, which contains correct and up-to-date R-stream memory state.

The only hardware support for this recovery method is a global invalidation control signal that resets the valid bit of all cache lines. Also, whereas the recovery controller is slightly imperfect due to the *persistent-stale* case, flushing the cache is completely effective.

3.2.3 Flush dirty lines

Invalidating all lines in the A-stream L1 cache is inefficient because only a handful of memory locations are typically corrupted. There are undue compulsory misses after recovery. To reduce A-stream compulsory misses, we propose invalidating only *dirty* lines. Any data written to the cache after the A-stream goes astray may be incorrect. And even if data is written correctly, our recovery strategy requires stores not yet performed in the R-stream to be “undone” in the A-stream at the time of recovery (to re-synchronize state). Thus, lines that become dirty after the A-stream diverges are good candidates for invalidating. Unfortunately, lines that were dirty before the A-stream diverged and not subsequently written to are needlessly invalidated.

As with the recovery controller, flushing dirty lines leads to imperfect recovery. First, stale lines brought in from the L2 cache that are not subsequently written to (clean) will persist after recovery. So, flushing dirty lines also suffers the *persistent-stale* problem. Second, clean lines that are corrupt due to incorrectly-skipped stores persist after recovery (the recovery controller, on the other hand, tracks addresses of correctly- and incorrectly-skipped stores). In Section 5, we measure how often bytes corrupted by a skipped-store before recovery are referenced by the A-stream after recovery; we label this measurement *persistent-skipped-write* in Section 5.

The hardware support for this recovery method is 1) dirty bits in the L1 cache and 2) a global invalidation control signal gated by the dirty bit that resets the valid bit of dirty cache lines.

3.2.4 Reducing impact of flush-induced misses: value prediction using preserved cache data

For either recovery method in Sections 3.2.2 and 3.2.3, flushing a cache line resets its valid bit but preserves its tag and data. The preserved tag and data can be exploited to reduce the impact of recovery-induced compulsory misses in the A-stream. When the A-stream accesses an invalid line, the cache miss is serviced like usual. However, the preserved cache tag(s) are still checked and, if there is a match, the A-stream retrieves a value from the cache and uses it as a value prediction. The value prediction is eventually validated when the cache miss completes.

Value predicting a load miss in this way gives some performance benefit, even if the load reaches the head of the reorder buffer and stalls A-stream retirement while waiting for the cache miss to complete. First, execution of dependent instructions is not delayed and this results in a faster retirement rate when retirement eventually resumes. Second, the load is unlikely to stall retirement for too long, if at all, because the invalidated line is likely to be in the L2 cache if it was found in the L1 cache. Third, other recovery-induced misses potentially initiate earlier. Finally, value predictions are nearly 100% accurate because, typically, only a handful of lines are corrupted when the A-stream deviates.

3.3 Simplified slipstream microarchitecture

Figure 5 shows the slipstream microarchitecture using hardware-based memory duplication and either of the flush-based recovery models. This microarchitecture does not need a recovery controller, so now there are only three slipstream components — the IR-predictor, IR-detector, and delay buffer. The L2 cache is shown. Notice the A-stream only reads from the L2 cache and the R-stream reads and writes the L2 cache.

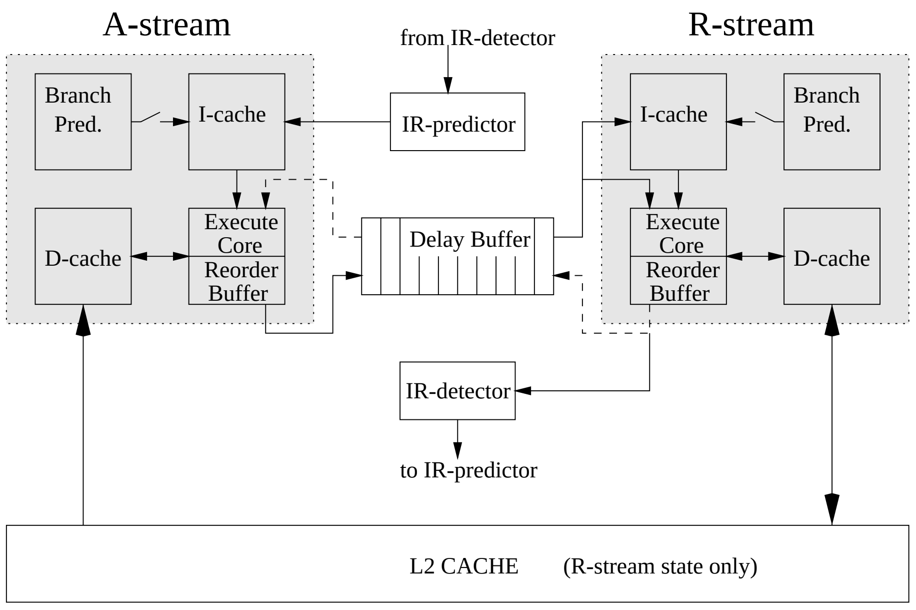

FIGURE 5. Slipstream microarchitecture with hardware-based memory duplication and flush-based recovery.

3.4 Summary: qualitative comparisons

Table 1 recaps the advantages, disadvantages, and required hardware support of the two memory duplication methods (top-half) and three memory recovery methods (bottom-half). Notice the four useful measurements introduced in this section are highlighted in bold italics: *stale* and *self-repair* relate to memory duplication; *persistent-stale* and *persistent-skipped-write* relate to recovery. Results in Section 5 quantify much of the information summarized in Table 1.

Note that the cache-based value prediction technique is not listed in Table 1, but is used in conjunction with either flush model to reduce the performance impact of flush-induced misses.

TABLE 1. Comparisons of duplication and recovery methods.

		POSITIVES	NEGATIVES	HARDWARE SUPPORT
memory duplication method	software-based	+ simple state renaming	- double memory usage - requires recovery controller - hard system-level issues	none
	hardware-based	+ as simple as s/w-based + efficient memory usage + enables simpler recovery + system-transparent + ***self-repair***	- ***stale***	No explicit hardware mechanisms, only assumes IBM-Power4-like CMP memory hierarchy
memory recovery method	recovery controller	+ restore data without flushing line from cache	- adds h/w complexity - explicitly increases recovery latency - force recovery when full - imperfect recovery: ***persistent-stale***	recovery controller mechanism
	flush	+ simple + 100% recovery	- many compulsory misses	invalidate wire
	flush dirty lines	+ simple + flush fewer lines	- some compulsory misses - imperfect recovery: ***persistent-stale*** ***persistent-skipped-write***	invalidate wire, dirty bits

4. Simulation methodology

We use a detailed execution-driven simulator of a slipstream processor. The simulator faithfully models the architecture depicted in Figure 1 and outlined in Section 2: the A-stream produces real, possibly incorrect values/addresses and branch outcomes, the R-stream checks the A-stream and initiates recovery actions, A-stream state is recovered from the R-stream state, etc. The simulator itself is validated via a functional simulator run independently and in parallel with the detailed timing simulator [24]. The functional simulator checks retired R-stream control flow and data flow outcomes.

Microarchitecture parameters are listed in Table 2. The top-left portion of the table lists parameters for individual processors within a CMP. The bottom-left portion describes the four slipstream components. We use the same parameters as in previous work [19], to which the reader is referred for more details, and focus on the slipstream memory hierarchy (right-hand side).

TABLE 2. Microarchitecture configuration.

single processor core (PE)	
caches	private L1 instr. cache (*see memory hier. column*)
	private L1 data cache (*see memory hier. column*)
superscalar core	reorder buffer: 64 entries
	dispatch/issue/retire bandwidth: 4 instr/cycle
	4 fully-symmetric functional units
	4 loads/stores per cycle
execution latencies	address generation = 1 cycle
	load access = 2 cycles (hit)
	integer ALU ops = 1 cycle
	complex ops = MIPS R10000 latencies

slipstream components	
IR-predictor	2^{20} entries, *gshare*-indexed (16 bits branch history)
	block size = 16, 16 confidence counters per entry
	confidence threshold = 32
IR-detector	R-DFG = 256 instructions, unpartitioned
delay buffer	data flow buffer: 256 instruction entries
	control flow buffer: 4K branch predictions
recovery controller	128 entries, fully associative
	recovery latency (*after* IR-misprediction detected):
	• 5 cycles to start up recovery pipeline
	• 4 reg. restores/cycle (64 regs performed 1st)
	• 4 mem. restores/cycle (mem performed 2nd)
	• ∴ min. latency (no memory) = 21 cycles

slipstream memory hierarchy	
L1 instruction cache (per PE)	size = 64 KB
	assoc = 4-way
	replacement = LRU
	line size = 64 bytes
L1 data cache (per PE)	size = 8 KB / 32 KB / 64 KB
	assoc = 1-way / 4-way
	replacement = LRU
	line size = 64 bytes
L2 cache	unified instr./data
	shared among PEs
	size = 256 KB
	assoc = 4-way
	replacement = LRU
	line size = 64 bytes
	write-back policy
memory access times	L1 instruction hit = 1 cycle
	L1 data hit = 2 cycles
	L1 miss/L2 hit = 12 cycles
	L1 miss/L2 miss = 70 cycles
# out. misses	unlimited for all caches
DUPLICATION	software- or hardware-based
RECOVERY	recovery controller, flush, flush-dirty, or flush/flush-dirty with value prediction

The per-PE L1 data cache size/associativity is varied. The shared L2 cache is 256KB, 4-way set-associative, and holds both instructions and data. Instruction pages are read-only so they are not duplicated even if software-based duplication is used. We also reduce A-stream/R-stream conflicts in the L2 cache for software-based duplication by inverting the high index bit for R-stream accesses (otherwise, the two address streams are too alike). An L1 data cache hit is 2 cycles, an L1 miss/L2 hit takes 12 cycles, and round-trip time to main memory is a minimum of 70 cycles.

Finally, we vary the memory duplication method and recovery method. The recovery controller (if present) holds 128 addresses and is fully-associative. Independent of the memory recovery method, recovery latency (*after* the IR-misprediction is detected) is 5 cycles to startup the recovery pipeline followed by 4 register restores per cycle (a total of 21 cycles). An additional latency of 4 memory restores per cycle is incurred *if the recovery controller is used*. For flush-based recovery, we assume the global invalidation signal can flush the cache within a few cycles and is hidden by the 21 cycle register file recovery latency.

The Simplescalar [4] compiler and ISA are used. We use six of the SPEC2000 integer benchmarks, compiled with -O3 optimization, and run with *ref* input datasets (Table 3). The first billion instructions are skipped, and then 100 million instructions are simulated.

TABLE 3. Benchmarks.

benchmark	ref input dataset
gap	gap -l./ -q -m 8M ref.in
gcc	cc1 expr.i -o expr.s (note: SPEC2K version of cc1 is hardwired to -O3 optimization)
parser	parser 2.1.dict -batch < ref.in
perl	perlbmk -I./lib splitmail.pl 850 5 19 18 1500
vortex	vortex bendian1.raw
vpr	vpr net.in arch.in place.out dum.out -nodisp -place_only -init_t 5 -exit_t 0.005 -alpha_t 0.9412 -inner_num 2

5. Results

The slipstream processor is built on top of a CMP composed of two superscalar cores. In the slipstream paradigm, the goal is to leverage a second, otherwise unused superscalar core in a CMP to improve single-program performance. Therefore, all results are reported as the speedup of slipstreaming using two superscalar cores with respect to conventional execution on one of the cores. Except for L1 data cache size and set-associativity, the core is fixed and is called the BASE configuration: BASE is a 4-way issue dynamically scheduled superscalar processor with a 64-instruction reorder buffer.

We first compare software-based and hardware-based memory duplication (Section 5.1), demonstrating that the hardware-based approach is required for materializing slipstream performance. We then investigate five recovery models within the context of hardware-based memory duplication (Section 5.2). In all, we simulate six slipstream processor configurations, labeled with the memory duplication method — SD for (s)oftware-based (d)uplication and HD for (h)ardware-based (d)uplication — followed by the recovery model in parentheses — “rc” = recovery controller, “flush” = flush entire cache, “flush-vp” = flush entire cache and use as value predictions, “flushd” = flush dirty lines in cache, “flushd-vp” = flush dirty lines in cache and use as value predictions. The six configurations are SD(rc), HD(rc), HD(flush), HD(flush-vp), HD(flushd), and HD(flushd-vp). Recall that the recovery controller is the only valid recovery method for software-based memory duplication.

5.1 Software-based vs. hardware-based memory duplication

The instructions-per-cycle (IPC) performance improvement of SD(rc) and HD(rc) with respect to BASE are shown in Figure 6. There is one graph per benchmark and L1 data cache configuration is varied along the x-axis (for example, 8k-1 is an 8KB direct mapped cache, 32k-4 is a 32KB 4-way set-associative cache).

Based on the results in Figure 6, efficient memory duplication is required to materialize slipstream performance. HD(rc) almost always outperforms SD(rc), and by large margins in all benchmarks except *gap* and *perl*. The SPEC2000 benchmarks place reasonable stress on the memory hierarchy. Consequently, SD(rc) takes a large performance hit because it doubles the number of physical pages competing for the already highly-utilized memory hierarchy.

The performance impact of full duplication in *parser* and *vpr* is large enough to degrade performance with respect to BASE by up to 5%. On the other hand, HD(rc) improves performance by about 17% in *parser* and 7% *vpr.* SD(rc) improves performance by about 8% and 5% in *gcc* and *vortex*, respectively. However, HD(rc) is able to increase those speedups to as high as 14% and 20%, respectively.

The *gcc* benchmark shows interesting trends for HD(rc) as cache size and set-associativity are increased. *Gap*, *perl*, and *vortex* also show these trends, but less clearly. First, performance improvement increases steadily with successively larger direct mapped caches. Yet, performance improvement is constant with cache size if the cache is 4-way set-associative. Second, there is a jump in performance improvement when associativity is increased from direct mapped to 4-way.

These trends can be explained by examining the number of *stale* bytes referenced by HD(rc), shown in Figure 8 averaged across all benchmarks. The number of referenced *stale* bytes is virtually non-existent (< 300 bytes) for set-associative caches. In contrast, direct mapped caches result in a high number of *stale* bytes referenced (e.g., 55,000 bytes for 8 KB direct mapped). A direct mapped cache usually has more conflict misses than a set-associative cache. Conflict misses result in many evicted-and-lost line updates that are re-accessed in the L2 cache too soon, before the R-stream has a chance to re-create the lost data (resulting in additional, costly A-stream mispredictions). Evictions in set-associative caches are more likely to be caused by capacity misses than conflict misses, in which case the evicted-and-lost line update is less likely to be re-accessed soon. We conjecture that an A-stream victim cache [11] will improve HD(rc) performance improvement with direct mapped caches. The same analysis explains why HD(rc) performance improvement is sensitive to L1 cache size for direct mapped caches and not for 4-way set-associative caches. The *stale* problem always exists, but decreases, as direct mapped cache size is increased.

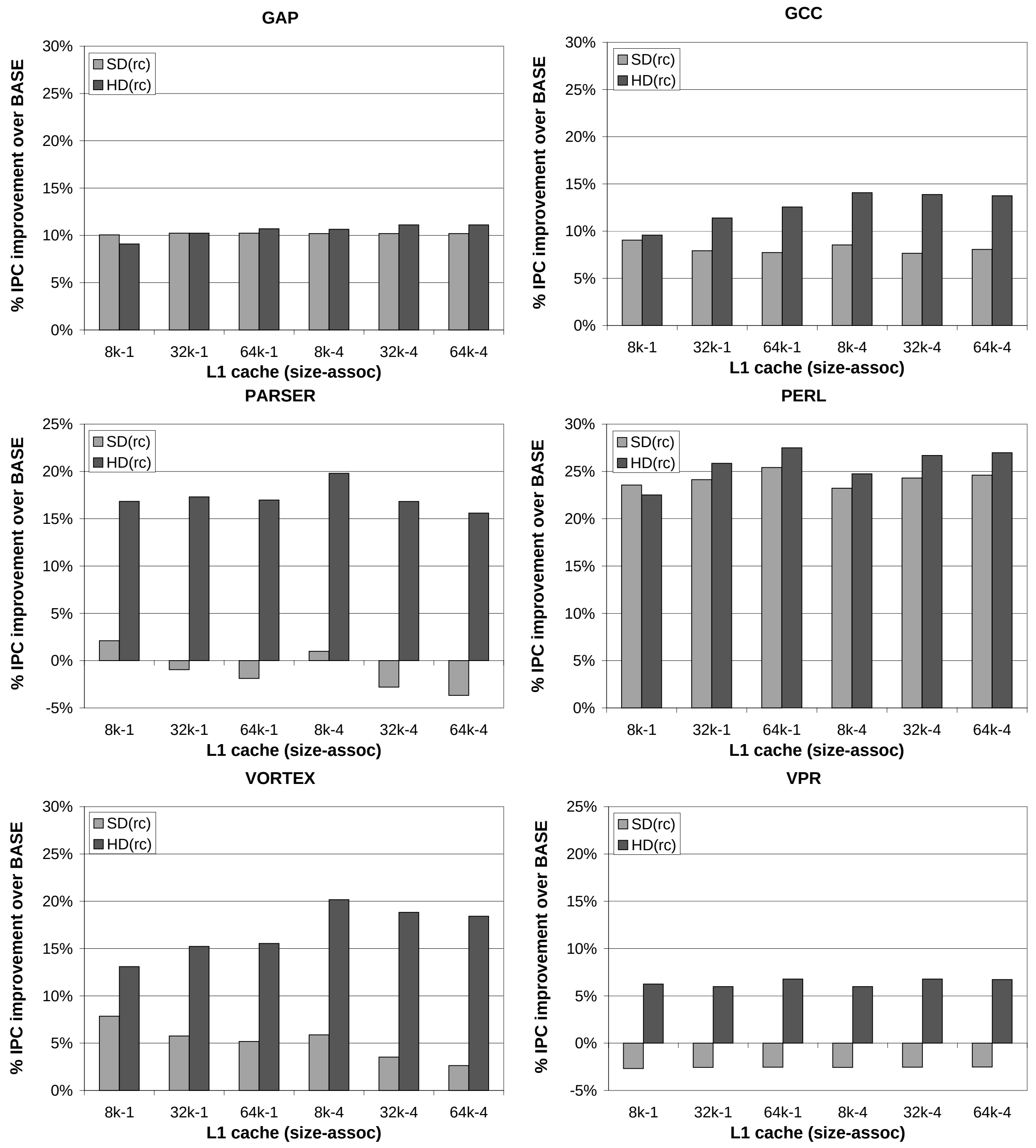

FIGURE 6. Comparison of duplication methods: performance of SD(rc) and HD(rc) with respect to BASE.

In Figure 8, the number of *self-repair* bytes referenced shows exactly the same trend as *stale* bytes referenced, i.e., direct mapped caches exhibit a lot of *self-repair*. This is to be expected, since conflict misses can actually be beneficial in terms of evicting corrupt lines before they are referenced.

5.2 Recovery model results

The IPC performance improvement of HD(flush), HD(flush-vp), HD(flushd), HD(flushd-vp), and HD(rc) with respect to BASE are shown in Figure 7, averaged across all the benchmarks. As before, L1 data cache configuration is varied along the x-axis.

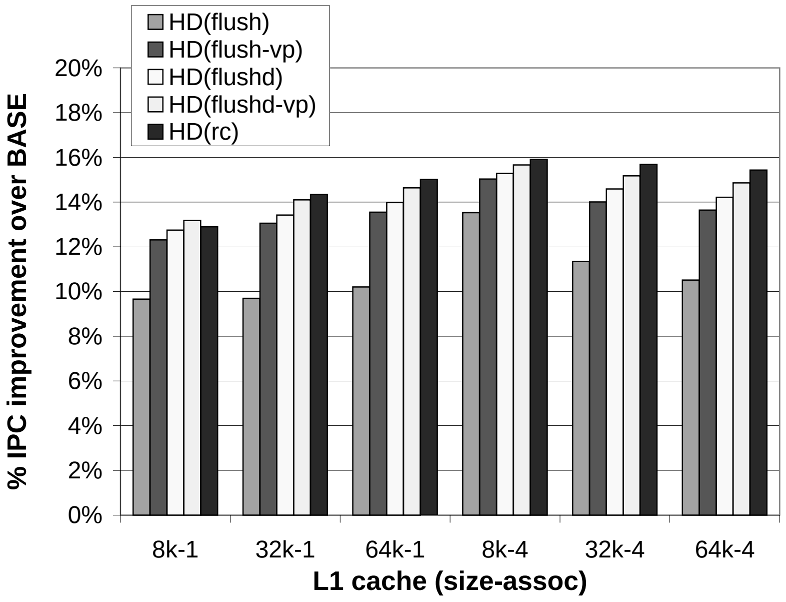

FIGURE 7. Comparison of recovery methods for hardware-based memory duplication, averaged across all benchmarks.

From Figure 7, on average, HD(rc) almost always performs best. This is to be expected because the recovery controller is both accurate and efficient, by virtue of pin-pointing corrupt data words.

The reason HD(rc) slightly underperforms HD(flushd-vp) for the 8KB direct mapped cache has to do with references to *persistent-stale* bytes. HD(rc) and HD(flushd) do not recover perfectly. Neither explicitly identifies stale cache lines, ultimately leading to *persistent-stale* data that remains after recovery. From Figure 8, the number of *persistent-stale* bytes referenced in HD(rc) is significantly higher than *persistent-stale* bytes referenced in HD(flushd). This is to be expected, because the less efficient flushing method inadvertently flushes some *stale* data, preventing *persistent-stale* data. This factor, combined with the value prediction enhancement to reduce flushing's cache miss penalty, pushes HD(flushd-vp) slightly ahead of HD(rc). This is only true for the 8KB direct mapped cache (also supported by data in Figure 8).

HD(flush) significantly underperforms the other recovery models, because of too many compulsory misses after recovery. HD(flushd) performs significantly better than HD(flush) because it flushes fewer cache lines. For a 32KB 4-way set-associative cache, HD(flush) drops slipstream performance improvement from 16% to 11%, whereas HD(flushd) only drops it down to 14.5%.

Using flushed data as value predictions significantly reduces the impact of recovery-induced misses. HD(flush-vp) performs close to HD(flushd) — 14% versus 14.5%, respectively, for the 32KB 4-way set-associative cache. And HD(flushd-vp) nearly closes the gap between HD(flushd) and HD(rc). For all cache configurations, HD(flushd-vp) is within a single percentage point of HD(rc). The significant result is that HD(flushd-vp) renders the recovery controller obsolete. In fact, all of the flush models except HD(flush) are effective alternatives to the recovery controller.

As discussed in the previous section, performance improvement of all HD(*) models is sensitive to direct mapped cache size due to *stale* data. This trend is observed again in Figure 7.

Performance improvement of HD(rc) is insensitive to cache size for the 4-way set-associative caches. Interestingly, the performance improvement of all of the flush models with respect to BASE decreases as the size of the 4-way set-associative cache is increased. The reason is the BASE processor benefits fully from the increased cache capacity, whereas the flush models do not benefit fully because lines are flushed during recovery. And the reason this trend was not visible for direct mapped caches is the *stale* problem dominates in that context.

As mentioned earlier, HD(rc) and HD(flushd) are imperfect recovery models. Figure 8 shows the *persistent-stale* problem is minor for both models, an order of magnitude smaller than the number of *stale* bytes referenced. HD(flushd) has the *persistent-skipped-write* problem as well. The *persistent-skipped-write* problem is also minor, fewer than 2,000 *persistent-skipped-write* bytes referenced over all cache configurations.

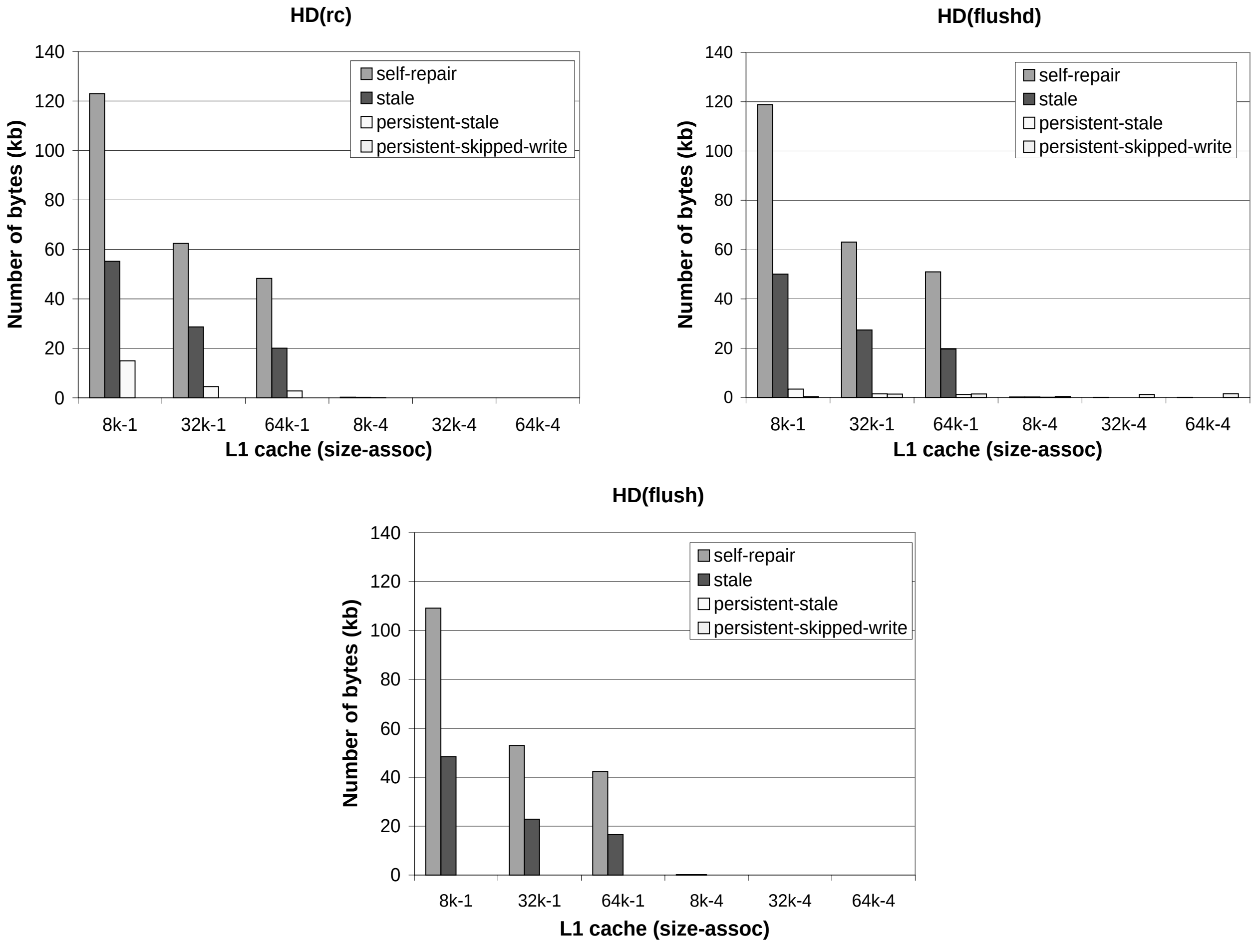

FIGURE 8. Number of referenced *stale, self-repair, persistent-stale,* and *persistent-skipped-write* bytes.

6. Related work

Speculative multithreading architectures [e.g.,1,13,17,24,26] speed up a sequential program by dividing it into speculatively-parallel tasks, and concurrently running the tasks on distributed processing elements or a simultaneous multithreaded pipeline. There is only one architectural context. Multiple speculative versions of a memory location can be created and hardware must explicitly track their program order. Version ordering is required because a dependent load must obtain the most recent prior version, and also because versions must be committed to the single architectural context in program order. Finally, loads may issue speculatively before prior dependent stores, so the versioning hardware also detects load violations after the fact.

The idea of leveraging private L1 caches in a CMP for renaming memory locations has been previously proposed in the context speculative multithreading, for example, SVC [10], TLDS [26], and MDT [13]. However, slipstreaming uses redundant programs instead of parallel tasks and this leads to certain simplifications. The A-stream and R-stream are functionally-complete programs with independent contexts — certainly with full duplication, but *also with constrained A-stream renaming storage*. Constraining the A-stream is only done for efficiency and this results in an *artificial* dependence on the R-stream, in which we rely on the R-stream catching up to the A-stream to reproduce lost A-stream data. Because conceptually there are no true dependences (only this new sort of artificial dependence), explicit mechanisms for ordering versions are non-existent, whereas ordering mechanisms are fundamentally required for parallel tasks.

Both slipstream and speculative multithreading demonstrate load speculation, but in fundamentally different ways. First, load misspeculation in the context of this paper is caused by limited A-stream storage, as opposed to ambiguous store-load dependences among tasks. Second, the R-stream is a general checking mechanism for verifying the forward progress of the A-stream [19,21], and specific misprediction-detection hardware is not required.

Finally, unlike SVC, TLDS, or MDT, the A-stream neither stalls nor initiates recovery when a cache replacement is needed. It simply loses data, implicitly predicting the R-stream will re-create the data before it is needed again. This may avoid many unnecessary stalls and recovery actions.

Speculative Data-Driven Multithreading [23] and related work [2,3,5,6,7,9,15,25,32], which spawn specialized threads to prefetch cache misses and resolve branch mispredictions in advance, are closer in spirit to slipstreaming. A fundamental difference is the use of multiple, short-lived, specialized threads versus a single, persistent, functionally-complete program (A-stream). This difference results in very different microarchitectures and, specifically, memory renaming has evolved differently. Use of the memory hierarchy (e.g., L1 cache or full duplication) is tailored towards the A-stream's persistence/completeness. Linking stores directly to loads via an explicitly-managed memory cloaking table [16], bypassing the memory system entirely, is tailored towards short-lived dependence-chain-based threads.

7. Summary

Slipstream processors use redundant program execution to speed up a single program, harnessing an otherwise unused PE in a CMP or thread context in an SMT processor. Full duplication of physical memory pages in software leads to a simple execution model due to program independence. Memory usage is doubled, however, and we showed this partially or fully negates slipstream performance benefits when a realistic memory hierarchy is simulated. Moreover, in the future we would like to investigate the potential memory latency tolerance benefits of slipstreaming, but inefficient duplication obscures that effort.

We showed it is possible to duplicate memory efficiently in hardware, without the complications normally associated with managing a fixed amount of rename storage in hardware. Representative CMP hierarchies (private L1 caches that write-through to a shared L2 cache) and the unique nature of slipstreaming are the sources of simplification. First, the already-replicated L1 caches in a CMP provide enough implicit rename storage. Second, this storage does not need to be explicitly managed because the slipstream paradigm is tolerant of slightly inaccurate memory renaming. Evicted L1 cache lines containing A-stream updates are simply lost, but the R-stream usually reproduces the lost data (which reaches the L2 cache via write-though) before the A-stream re-references the line in the L2 cache. And occasional references to stale data are not a problem because the A-stream is speculative in any case (furthermore, no special checking is required because the R-stream generally checks A-stream forward progress). In summary, the new hardware-based memory duplication requires no explicit rename storage nor explicit management, and significantly outperforms software-based memory duplication.

Another nice feature of hardware-based duplication is it enables much simpler state recovery. The A-stream can be re-synchronized to the R-stream by flushing the A-stream L1 cache. We showed compulsory misses after recovery limit performance, and that flushing dirty lines (while not 100% effective at restoring state) performs much better. And, using preserved data within flushed cache lines as value predictions allows the flush-dirty-line recovery model to perform within a few percent of the recovery controller, rendering that slipstream component obsolete.

8. Acknowledgments

This research was supported by generous funding and equipment donations from Intel, Ericsson, and by NSF CAREER grant No. CCR-0092832.

9. References

[1] H. Akkary and M. Driscoll. A Dynamic Multithreading Processor. *31st Int'l Symp. on Microarch.*, Dec. 1998.

[2] M. Annavaram, J. Patel, and E. Davidson. Data Prefetching by Dependence Graph Precomputation. *28th Int'l Symp. on Computer Architecture*, July 2001.

[3] R. Balasubramonian, S. Dwarkadas, and D. Albonesi. Dynamically Allocating Processor Resources Between Nearby and Distant ILP. *28th Int'l Symp. on Computer Architecture*, July 2001.

[4] D. Burger, T. Austin, and S. Bennett. Evaluating Future Microprocessors: The Simplescalar Toolset. Technical Report CS-TR-96-1308, Computer Sciences Department, University of Wisconsin - Madison, July 1996.

[5] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt. Simultaneous Subordinate Microthreading (SSMT). *26th Int'l Symp. on Computer Architecture*, May 1999.

[6] J. Collins, H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, J. Shen. Speculative Precomputation: Long-range Prefetching of Delinquent Loads. *28th Int'l Symp. on Computer Architecture*, July 2001.

[7] J. Dundas and T. Mudge. Improving Data Cache Performance by Pre-executing Instructions Under a Cache Miss. *Proceedings of ICS*, 1997.

[8] J. Emer. Simultaneous Multithreading: Multiplying Alpha Performance. *Microprocessor Forum*, October 1999.

[9] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow Analysis of Branch Mispredictions and its Application to Early Resolution of Branch Outcomes. *31st Int'l Symp. on Microarchitecture*, Dec. 1998.

[10] S. Gopal, T. N. Vijaykumar, J. E. Smith, and G. S. Sohi. Speculative Versioning Cache. *4th Int'l Symposium on High-Performance Computer Architecture*, February 1998.

[11] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. *17th Int'l Symp. on Computer Architecture*, May 1990.

[12] J. Kahle. Power4: A Dual-CPU Processor Chip. *Microprocessor Forum*, October 1999.

[13] V. Krishnan and J. Torrellas. Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-Multiprocessor. *1998 Int'l Conference on Supercomputing*, July 1998.

[14] M. Lipasti. Value Locality and Speculative Execution. Ph.D. Thesis, Carnegie Mellon University, April 1997.

[15] C.-K. Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. *28th Int'l Symp. on Computer Architecture*, July 2001.

[16] A. Moshovos and G. S. Sohi. Streamlining Inter-Operation Memory Communication via Data Dependence Prediction, *30th Int'l Symp. on Microarchitecture*, Dec. 1997.

[17] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K.-Y. Chang. The Case for a Single-Chip Multiprocessor. *7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[18] J. Oplinger, D. Heine, S.-W. Liao, B. Nayfeh, M. Lam, and K. Olukotun. Software and Hardware for Exploiting Speculative Parallelism in Multiprocessors. CSL-TR-97-715, Stanford University, Feb. 1997.

[19] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A Study of Slipstream Processors. *33rd Int'l Symposium on Microarchitecture*, Dec. 2000.

[20] S. Reinhardt and S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. *27th Int'l Symp. on Computer Architecture*, June 2000.

[21] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. *29th Int'l Symp. on Fault-Tolerant Computing*, June 1999.

[22] E. Rotenberg. Exploiting Large Ineffectual Instruction Sequences. Technical Report, Department of Electrical and Computer Engineering, North Carolina State University, Nov. 1999.

[23] A. Roth and G. Sohi. Speculative Data-Driven Multithreading. Technical Report CS-TR-2000-1414, Computer Sciences Department, University of Wisconsin - Madison, April 2000.

[24] G. Sohi, S. Breach, and T. N. Vijaykumar. Multiscalar Processors. *22nd Int'l Symp. on Computer Architecture*, June 1995.

[25] Y. H. Song and M. Dubois. Assisted Execution. Technical Report CENG-98-25, Department of EE-Systems, University of Southern California, October 1998.

[26] J. Steffan and T. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. *4th Int'l Symp. on High-Performance Computer Architecture*, Feb. 1998.

[27] S. Storino and J. Borkenhagen. A Multi-Threaded 64-bit PowerPC Commercial RISC Processor Design. *11th Hot Chips Symposium*, August 1999.

[28] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. *9th Int'l Conf. on Arch. Support for Programming Languages and Operating Systems*, Nov. 2000.

[29] M. Tremblay. MAJC: Microprocessor Architecture for Java Computing. *11th Hot Chips Symposium*, Aug. 1999.

[30] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *22nd Int'l Symp. on Computer Architecture*, June 1995.

[31] W. Yamamoto and M. Nemirovsky. Increasing Superscalar Performance through Multistreaming. *Parallel Architectures and Compilation Techniques*, June 1995.

[32] C. Zilles and G. Sohi. Execution-based Prediction Using Speculative Slices. *28th Int'l Symp. on Computer Architecture*, July 2001.