In Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA 2011)

Fast Thread Migration via Cache Working Set Prediction

Jeffery A. Brown   Leo Porter   Dean M. Tullsen
University of California, San Diego
La Jolla, CA 92093-0404

Abstract

The most significant source of lost performance when a thread migrates between cores is the loss of cache state. A significant boost in post-migration performance is possible if the cache working set can be moved, proactively, with the thread.

This work accelerates thread startup performance after migration by predicting and prefetching the working set of the application into the new cache. It shows that simply moving cache state performs poorly, and that moving the instruction working set can be even more critical than data. This paper demonstrates a technique that captures the access behavior of a thread, summarizes that behavior into a compact form for transfer between cores, and then prefetches appropriate data into the new caches based on the summary. It presents a detailed study of single-thread migration effects, and then demonstrates its utility on a speculative multithreading architecture.

Working set prediction as much as doubles the performance of short-lived threads, and in a full speculative multithreading implementation, the technique is also shown to nearly double the effectiveness of the spawned threads.

1. Introduction

As we progress into the manycore era, we become increasingly dependent on high levels of thread-level parallelism for performance scaling. This will require new programming and execution models that expose more parallelism. An important barrier to the viability of many proposed execution models is an inability to efficiently execute short threads, due to the overhead of copying a thread's cache state between cores. Several new execution models significantly decrease the mean core occupancy times of threads, or otherwise increase the frequency of thread state transfers between cores.

Such new models will be significantly more effective if they have the freedom to exploit parallelism in chunks which are tens to hundreds of instructions long. However, current machines cannot profitably move or fork execution between cores at ranges below tens to hundreds of *thousands* of instructions; a great deal of potential parallelism is unavailable due to the cost of moving and forking threads.

We also see frequent state migration with traditional parallelization, both at loop-level (where threads spawned for a loop iteration inherit the state of the serial code leading to the loop) and at task-level (where parallel tasks inherit the state of the callers). Less traditional uses of parallel hardware also demand frequent migration. Speculative multithreading [32, 14, 27, 34] breaks serial execution into potentially parallel threads, each thread inheriting the execution context of the previous thread. Helper threads [7, 11, 40] also utilize parallel hardware for speedup, without actually offloading computation; each new helper thread executes within the same address space as the main thread, inheriting its memory state.

Heterogeneous multi-core proposals [18, 19] move threads between cores to exploit power-performance-area trade-offs. Those proposals use frequent sampling, via heavy thread migration, to discover good mappings of threads to cores. They migrate threads conservatively due to the high cost of migration; presumably, as we lower that cost, such architectures can adapt more quickly, increasing potential gains.

Other research migrates threads when thread-level parallelism changes [1] or at each system call [22, 5]. Software data spreading [17] frequently migrates threads at compiler-determined points in order to effectively utilize the aggregate capacity of multiple private caches. Even when multi-cores are not exploited for performance, thread migration may be demanded, e.g. for schemes which use core-hopping for thermal management [6].

These techniques all share the property that a thread begins or resumes execution on one core after its working set has been built up on another core. In current systems, the primary mechanism for working set migration (WSM) is executing code on the new core, causing demand misses which retrieve data from either another core, shared cache, or memory. This is a particularly inefficient mechanism for building a working set, since the rate at which data migrates is directly tied to the speed of the "migration engine" — the executing code. The time when execution is most sensitive to cold-cache effects – just after migration – is the very time when execution generates addresses most slowly, because performance suffers due to those same effects.

We explore mechanisms for predicting and prefetching the future working set of threads as they migrate between cores. We introduce a three-step approach to WSM: first, we augment each core with simple hardware to *capture* the access behavior of threads as they execute. Next, when deactivating or forking a thread, we *summarize* the captured behavior to represent likely future instruction and data accesses, then transfer that summary along with other thread state. Finally, we *apply* the summary data with a prefetcher at the new core.

Our primary purpose is to evaluate a diverse set of mechanisms which capture access behavior, and measure how effectively each predicts future accesses. Though we evaluate schemes of varying complexity, we achieve our best results using very inexpensive schemes. We realize useful performance gains at low cost with small, low-complexity tables, maintained using only the address stream of executing threads.

We also demonstrate the utility of WSM by adding it to a speculative multithreading architecture, where it significantly boosts the effectiveness of speculative threads.

This research makes the following contributions: We show that demand-fetching is not an effective mechanism for filling caches after migration. We show that conventional hardware prefetchers are not useful over the time intervals in which performance loss is the most dire. We show that in many scenarios I-stream misses are much more critical to post-migration performance than D-stream misses. We demonstrate that bulk transferring the private caches is surprisingly ineffective, and in many cases is worse than doing nothing. With the addition of a few small, simple tables to monitor access activity, and a prefetcher driven by those tables, we achieve as much as a 2X performance boost for short (100-instruction) threads. We show that some of these techniques directly apply to the problem of lost locality due to frequent thread spawning in a speculative multithreading architecture.

2. Background and Related Work

Previous work [3, 38] describes support mechanisms for migrating register state in order to decrease the latency of thread activation and deactivation; however, performance subsequent to migration still suffers due to cold-cache effects. Our work is complimentary; we specifically address the post-migration cache misses which limit the gains of those techniques. Choi, et al., explore the complementary problem of branch prediction for short-lived threads [8].

Stream buffers [16, 24] introduce small associative structures which track data access patterns. Additional work [31] extends this idea, allowing an advanced predictor to be shared among many streams. We model a discrete hardware predictor-directed stream buffer in the style of [31], omitting the shared Markov predictor, as part of our baseline. We also add the ability to transfer stream-buffer state as a candidate working set prefetcher.

Sair, et al. [29] survey several prefetchers, and introduce a method for classifying memory access behaviors in hardware: the memory access stream is matched against behavior-specific tables operating in parallel. We utilize some similar structures in the *capture* stage of our migration system.

Speculative Precomputation [40, 11] targets memory instructions which degrade performance due to poor cache behavior, using alternate contexts on multithreaded or CMP [4] architectures. Focusing on misses, these schemes target the subset of the future working set which is not currently cached. Dependence-following schemes [2, 10] prefetch by following dependence chains through memory. While valuable, these techniques alone are of limited utility immediately after a migration, due to their serial progression. (We incorporate this style of prefetching with our *pointer-chase* table.)

Runahead Execution [23] prefetches by speculatively executing the application, but ignoring dependences on long-latency misses. This seems well-suited to our need to prefetch what would normally be cache hits in addition to misses, and to cover a substantial amount of working-set with little metadata overhead. However, this scheme is hamstrung in the post-migration environment by the lack of I-cache state at the target core; I-cache miss stalls serialize short-term prefetching.

Dead-block prediction [20] predicts when L1 D-cache blocks are no longer needed before replacement; an additional table is used to predict likely successors for early-evicted blocks. They use the dead-block predictor and correlations it exposes to prefetch likely misses, and use the freed L1 storage as a prefetch buffer. Their work motivates ours; we also find that caches often hold data irrelevant to future accesses.

Data Marshaling [36] mitigates inter-core data misses in Staged Execution models. In contrast to our approach, DM targets scheduled stage transitions using compile-time flagging of producer instructions, hardware to track writes by flagged instructions, and a new instruction which triggers data transfers.

3. Baseline Multicore Architecture

We study a four-core chip multiprocessor (CMP). Each core has a four-way superscalar out-of-order execution engine. Given their ability to exploit memory level parallelism, these cores will be *less* sensitive to cache migration effects than those of a conservative design. Our cores have private first-level I- and D-caches, and a private second-level unified cache; see Figure 1. Off-chip memory is accessed via a shared four-channel off-chip memory controller. Specific parameters of the core and memory subsystems are detailed in §6.1. The four cores communicate over a shared bus. Caches are kept coherent with a MESI

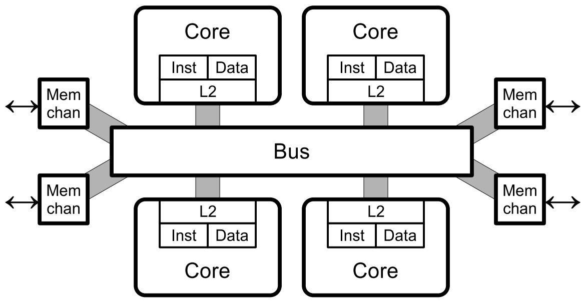

Figure 1: Baseline multicore processor

coherence protocol [25] and snooping; our techniques readily apply to systems with scalable interconnects and more cores.

The cores of our CMP feature hardware support for thread activation and deactivation, as found in prior studies of thread scheduling [3, 38]. While those works used hardware support to implement scheduling and time-sharing policies, we use it simply for adding and removing threads from cores. Traditional software-driven migration has much higher overhead, which would dominate cache migration costs; we believe that direct OS involvement in all thread movement is ceasing to be a viable model, but even the OS overhead for migration can be significantly reduced from current levels [35].

4. Motivation

Many execution scenarios require the working set of a thread to migrate between cores: load balancing, thread spawning, loop-level parallelization, task-level parallelization, helper threads, speculative multithreading, heterogeneous multicore adaptation, thermal management, etc. These will become more common as core counts increase; each of these mechanisms will be even more effective with decreased migration cost. Although the principles of this work apply in all of these cases, for clarity of evaluation we focus first on single-thread migration at arbitrary points in the program. We later apply our technique to a speculatively multithreaded, transaction-based parallel workload, improving overall performance.

To initially evaluate migration mechanisms on our single-thread workloads, we repeatedly move a single thread among a set of cores. There are several costs incurred in migrating a thread: transferring register state, transferring TLB state, recreating branch predictor state, etc.; however, the largest amount of program state on a core resides in the caches. As a result, the cost of transferring cached state dominates thread re-start performance. In addition, cache state is moved very slowly, because it is only demand-fetched as the thread executes on the new core, but after migration that thread is executing (and demand-fetching) extremely slowly.

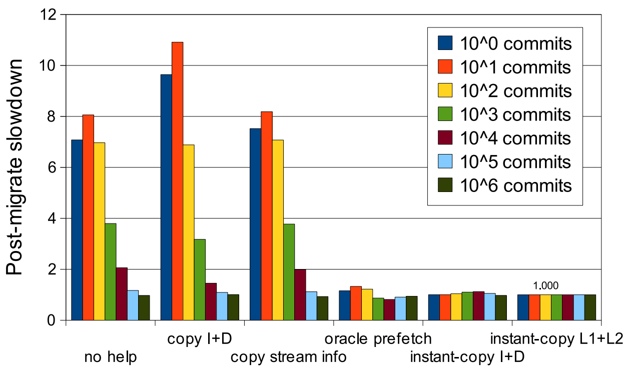

Figure 2: The cost of migration, in reduced instruction throughput, for various assumptions about the migration of data. The baseline is instant replication of all private caches.

Figure 2 gives the result of an experiment that illustrates the cost of migration and the potential to reduce the cache-related portion of that cost. We force a single thread to migrate round-robin among four cores, moving every 1 million commits, and record the time it takes to start-up and commit the next $1, 10, 100, \ldots 10^6$ instructions after each migration. We perform this experiment across the SPEC2000 benchmark suite, with varying amounts of architectural (and oracle) support for migration. We show slowdown relative to the ideal case where all cache contents are instantly transported to the new core for free. It takes, on average, 7 times as long to commit the 100th instruction in the default migration case – "no help" – compared to cost-free cache migration. (We initially assume that a background workload causes cache state to be evicted before a thread returns to a previous core, and which does not otherwise impair the thread; we revisit this assumption in §7.6.)

Considering the feasible schemes of Figure 2, "copy I+D" bulk-copies both first-level caches by transferring a list of their tags to the new core and fetching blocks via core-to-core transfers; "copy stream info" transfers just the metadata from the first core's hardware stream buffer to the second, allowing it to resume following known streams. For the idealized schemes, "oracle prefetch" uses perfect knowledge of future accesses to fetch required blocks at the new core, requesting them via the memory hierarchy as the thread restarts and modeling the costs of these requests; "instant-copy I+D" instantly transfers the contents of the first-level I- and D-caches to the new core, cost-free; finally, "instant-copy L1+L2" instantly transfers all cached data, cost-free.

This graph provides several key insights. First, we see that unless a thread executes on a core for many instructions before being migrated, the cost of migration is not amortized in the realistic schemes. At 10,000 commits, the cost is still very high (2X slowdown); for shorter threads, migration cost is extreme. Note that several speculative multithreading proposals routinely execute threads under 100 instructions [21], as do helper-threads [39]; several transactional memory programs (for Transactional Coherency and Consistency) showed average transaction lengths in the low hundreds [9].

We also see that copying entire caches proactively – the "copy I+D" case – is not effective: there is too much data, much of it irrelevant. This is worse than doing nothing over short intervals, and takes about 10,000 instructions to be amortized enough to approach break-even. While not shown, the cost of copying the larger L2-resident state is even higher.

We see that copying stream buffer state – the "copy stream info" case – is more effective over short terms than moving the entire L1 cache state; the stream buffer is small and directly targets future accesses. This is still not particularly effective, since it represents only a small fraction of the future working set: the stream buffer is built to target future *misses* in the D-stream; what we really want is something similar to the stream buffer, but trained on the entire access stream. Furthermore, this scheme does not prefetch for the I-stream. Immediately after migration, there is great demand for instructions; the lack of I-stream prefetching is exacerbated by the inability to overlap multiple I-misses.

At the short time scales we're most interested in for migration support, conventional hardware prefetchers are unable to contribute much (we observe this, because hardware prefetchers are included in our baseline architecture): they don't have enough time to train on an incoming thread's behavior, since the thread itself is struggling to execute.

5. Architectural Support for WSM

In §4 we demonstrated that characterizing the miss stream is not sufficient to cover many migration-related misses; we need to characterize the *access* stream. We construct a working set predictor which works in three stages. First, we observe the access stream of a thread and *capture* patterns and behaviors. Second, we *summarize* this behavior and transfer the summary data to the new core. Third, we *apply* the summary via a prefetch engine on the target core, to rapidly fill the caches in advance of the migrated thread.

We begin by describing a set of possible capture engines. We start out concerned less with the implementation cost, and more with exploring a variety of possibilities. After evaluating tables that capture a wide variety of access patterns, we find that we get excellent performance with very few, very small tables, each with simple index functions, utilizing only post-commit PCs and memory addresses.

5.1. Memory logger

To each core we add a *memory logger*, a specialized unit which records selected details of each committed memory instruction and I-cache access. This unit passively observes the current thread as it executes, but does not directly affect execution; it can be implemented outside of the main pipeline where it need not influence critical timing paths, and is tolerant of any additional latency needed for buffering. If a conservatively-implemented memory logger becomes swamped with input, records may safely be discarded; this will degrade the accuracy of later prefetching. We model a memory logger which can keep up with execution.

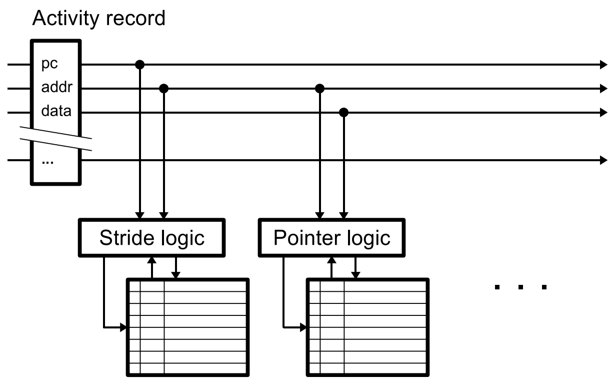

Figure 3: A memory logger implementation with multiple table types.

The memory logger is implemented with small (32-entry) content-addressable memory (CAM) tables, each with associated control logic. These tables are indexed using various portions of the information from each memory operation. We examine a variety of potential table types, each tailored to capture a specific class of access pattern: one table tracks striding accesses, another tracks pointer traversals, etc. Figure 3 shows an overview of a potential memory logger architecture with multiple tables. As mentioned, we find in §7.3 and §7.4 that we get excellent performance with a few simple tables, and very little information extracted from the pipeline.

Each table within the memory logger targets a specific type of access pattern; we next describe each type we consider. (We are not proposing these tables as novel prefetching schemes; several of the underlying ideas are discussed in §2, and [29] surveys several more.)

Next-block-{Inst,Data}: These detect sequential block accesses; entries are advanced by one cache block on a hit. We maintain separate next-block tables for the I- and D-streams.

StridePC: This tracks individual instructions which walk through memory in fixed-sized steps; only D-stream accesses are tracked, using the PC values and access addresses.

Pointer, Pointer-chase: These capture active pointers and pointer traversals, respectively, by detecting when the data output of a load matches the address of a later memory access, similar to pointer-cache [10] and dependence-

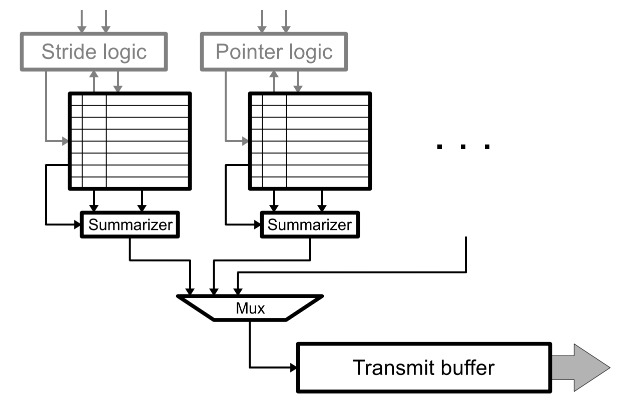

Figure 4: Summary generator

based [28] prefetching. *Pointer* tracks loaded values used as addresses without following them, while *Pointer-chase* replaces each entry with the target value when a load match is observed.

Same-object: This captures accesses to ranges of memory from a common base address, as is common for structure and object access code. This takes advantage of the common "base+offset" addressing mode, tracking minimum and maximum offsets for each base address, while ignoring accesses relative to the global pointer or stack pointer.

SPWindow, PCWindow: These don't actually use tables; we just record the value of the stack pointer and PC, respectively, and use them to prefetch a window of data blocks near the top of the stack, or a window of instructions near the first post-migrate instruction.

{*Inst,Data*}*-MRU:* These record the most recent blocks accessed from the I- and D-streams. These operate at four-cache-block granularity, allowing them to cheaply cover a larger number of blocks without increasing MRU maintenance.

BTB, BlockBTB: These capture taken branches and their targets, recording the most recent inbound branch for each target. *BlockBTB* is a block-aligned variant of *BTB*; branch and target PCs are block-aligned, allowing a greater amount of the instruction working set to be characterized at a given table size.

RetStack: This maintains a shadow copy of the processor return stack, prefetching blocks of instructions near the top few control frames.

5.2. Summary generator

The *summary generator*, depicted in Figure 4, activates when a core is signaled to migrate a thread. As introduced in §3, our baseline core design assumes hardware support for thread swapping; at halt-time, the core collects and stores the register state of the thread being halted.

While register state is being transferred, the summary generator reads through the tables populated by the memory logger and prepares a compact summary of the thread's

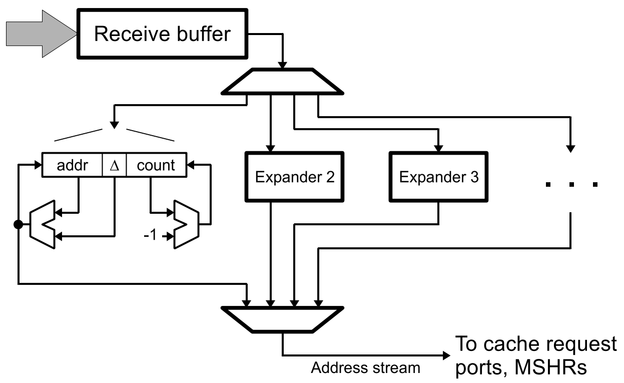

Figure 5: Summary-driven prefetcher

likely future working set. This summary is transmitted after the architected thread state, and is used to prefetch the thread's working set when it resumes on the new core. During summarization each table entry is inspected to determine its usefulness by observing whether its age-adjusted hit counter exceeds a threshold. Output summaries are packed into cache-line size blocks for efficient transfer.

Table entries are summarized for transfer by generating a sequence of block addresses from each, following the behavior pattern captured by that table (e.g. prefetching an object's blocks for a *Same-object* entry, or the successor block for a *Next-Block* entry). We encode each sequence with a simple linear-range encoding, <*start-address, stride, length*>, which tells the prefetcher to fetch *length* cache blocks, starting at *start-address*, with stride *stride*. We conservatively allocate 64 bits for each tuple; this could be reduced with even rudimentary compression. The *length* bound is necessary because we prefetch directly into the caches, dependence-free, as fast as the memory subsystem allows. This does not have the natural pacing present in, for example, stream buffers; while finite MSHR and cache ports limit overall prefetcher throughput, judicious *length* restrictions help individual summaries share those resources. Overall, we've tuned lengths to roughly transfer enough data to cover the first 1,000 instructions.

5.3. Summary-driven prefetcher

Rounding out our working-set migration hardware is the summary-driven prefetcher, depicted in Figure 5. When a previously-suspended thread is activated on a core, its summary records are read by the prefetcher. Each record is expanded to a sequence of cache block addresses, which are submitted for prefetching as bandwidth allows. While the main execution pipeline reloads register values and resumes execution, the prefetcher independently begins to prefetch a likely working set. Prefetches search the entire memory hierarchy, and contend for the same resources as demand requests. Once the register state has been loaded, the thread resumes execution and competes with the prefetchers for memory resources. Communication overlap and contention for interconnect and caches is modeled among transfers of register state, table summaries, prefetches, and the service of demand-misses.

We model the prefetch engine as submitting virtually-addressed memory requests at the existing ports of the I- and D-caches, utilizing the existing memory hierarchy for service. While these requests compete with the thread itself, we find that most prefetching immediately after migration occurs while the thread would otherwise be stalled for memory access; thus we do not require additional cache porting.

6. Methodology

We evaluate the prefetching coverage and overall performance of our working-set migration system using an execution-driven, out-of-order processor and memory system simulator.

6.1. Simulator configuration

We start with a multi-core version of SMTSIM [37], configured with single-threaded cores. Our four cores are clocked at 2.0 GHz, with timing for other structures reported in terms of this clock rate. Table 1 lists the most significant parameters of the baseline system used in our experiments. Our memory subsystem layout and latencies are based on that of recent Intel Nehalem-based processors [15]; bandwidth constraints are based on benchmarking of Nehalem-based systems and on the specifications of DDR3-1600.

Our system details MSHRs, queues, banking, and porting for each cache, with per-bank and port latency and bandwidth accounting. We model latency and bandwidth for each DRAM channel with a simple queue, "QILM" in the parlance of [33], without detailed memory controller modeling. In this study, virtually all post-migration misses are serviced core-to-core rather than off-chip; consequently, the memory hierarchy beyond the L2 caches has no significant impact on our results. (We confirm this in §7.8, where our results are unaffected by adding a shared L3 cache, and again in §8 where our overall speedups are unaffected by large changes in DRAM latency.)

Atop this baseline, we implement and evaluate the working-set migration architecture described in §5. The *InstMRU* and *DataMRU* CAMs have 16 entries each; all other logger CAMs have 32 entries. We use an easily-implemented round-robin replacement policy, which skips replacement of entries with recent hits (within the last 500 lookups), avoiding the complexity of LRU.

6.2. Workloads

Evaluating the speed of migration is not straightforward. We could examine a particular environment which benefits from fast migration (e.g. speculative multithreading, a shared-thread multiprocessor), but the results would be specific to those execution models; instead, we first model a generic environment to produce techniques that are useful in the general case. While there are many potential reasons for a particular migration (as discussed in §1), *any* migration in a real system will be subject to some or all of the overheads we characterize in this work.

Parameter	*Value*
Clock rate	2.0 GHz
Fetch width	4
Reorder buffer	128 entries
Integer window	64 insts
FP window	64 insts
Max issue width	4
Integer ALUs	4
FP ALUs	2
Load/store units	2
Branch predictor	4 Kbit gshare
BTB	256-entry, 4-way
Cache block size	32 B
Page size	8 KB
L1 I-cache size/assoc.	32 KB/4-way
L1 I-MSHR	16 entries, 32 waiters each
L1 D-cache size/assoc.	32 KB/4-way
L1 D-cache ports	2 read/write
L1 D-MSHR	16 entries, 32 waiters each
L2 cache size/assoc.	512KB/8-way, per-core
L2 cache ports	8 banks, 1 port ea.
ITLB entries	48
DTLB entries	128
Load-use latency, L1 hit	2 cyc
Load-use latency, L2 hit	14 cyc
Load-use latency, memory	176 cyc
Load-use latency, cross-core	34 cyc
TLB miss penalty	160 cyc
L1 I-cache ideal b/w	60 GB/s
L1 D-cache ideal b/w	60 GB/s
L2 cache ideal b/w	60 GB/s
Bus ideal b/w	30 GB/s
Mem ideal b/w	30 GB/s R, 20 GB/s W
Thread activate latency	15 cyc, to first fetch
Thread deactivate latency	44 cyc, to fetch available
L1 D stream buffer	8 streams, 4 blocks/stream
L1 D stream buffer stride table	256-entry, 4-way

Table 1: Baseline processor parameters

We start with all individual benchmarks from the SPEC2000 suite, running standalone on a four-core processor. Each benchmark is simulated for 200 million commits during its main phase of execution. To evaluate performance subsequent to migration, we force threads to migrate round-robin around cores, triggering a migration every 1 million commits. We measure the time to commit 10^n instructions after each migration, $n \in \{0 \dots 6\}$. Examining performance across this wide range of intervals offers insight into how long it takes to amortize the cost of a migration, and the expected throughput for short, medium, and long-lived threads.

We initially assume that caches are empty when a migrating thread returns to a given core. This is a simplified model of an environment where unrelated background threads occupy other cores, evicting the blocks of the thread under study during its absence; this allows evaluation absent noise from other threads. We also present data from more realistic scenarios with background threads simulated in §7.6 and §7.7. In those cases, we use random sets of the other benchmarks running on the idle cores, with threads on the other cores migrating among cores about as often as the thread under measurement.

In addition to the detailed evaluation of post-migrate behavior described in this section, we also demonstrate the overall benefit of working-set migration on a particular architecture, speculative multithreading. Because those experiments introduce several new properties, that methodology is discussed in §8.

6.3. Metrics

Performance evaluation presents another challenge: we wish to gauge the impact of our changes on performance in the immediate wake of migration operations, which we repeatedly induce. By triggering fairly infrequently and capturing the post-migration behavior over a wide range of intervals, we capture both the short-term and long-term impact of each migration. Our measured migrations are independent of any particular system condition other than overall commit progress; this ensures that the results at each time interval are comparable across various migration policies, since migrations occur at the exact same points in each simulation.

Our performance metric is based on the time it takes to commit an interval of 10^n instructions immediately following each migration operation, for $n \in \{0 \ldots 6\}$. We measure time from the first post-migrate fetch until 10^n instructions commit. We report results in terms of speedups relative to the same intervals on our baseline system.

7. Analysis and Results

This section examines a number of mechanisms for predicting the future working set of a migrating thread, ranging from simple to complex, ideal to realistic.

7.1. Bulk cache transfer

The most straightforward predictor of the future working set is the existing contents of the caches. We can copy those contents in bulk immediately after moving register state. We saw in Figure 2 that this was not effective, at least over the short intervals, due to the quantity of data transferred and the fraction of data that does not turn out to be useful. To perform bulk cache transfers at migration time – while the source core's pipeline is retrieving register values for transfer – we read out the set of cache tags belonging to the subject thread, and pack them into a message for transfer to the target core.

Figure 6 shows the impact of bulk cache transfers on our single-threaded workloads, broken down by individual

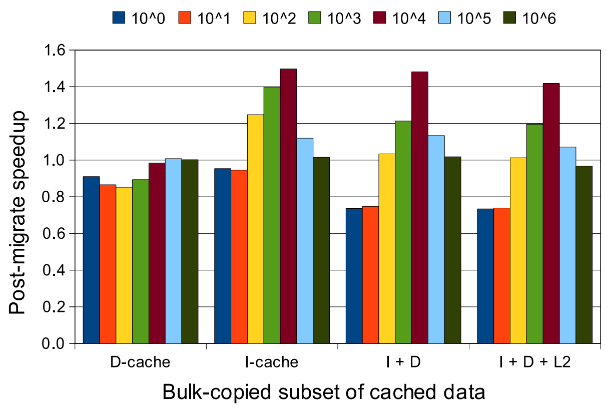

Figure 6: Impact of adding bulk cache transfers to single-threaded workloads. Speedups are relative to no migration support.

cache. Over the shorter intervals, performance is significantly worse, as the dependence-free prefetch traffic consumes most available request bandwidth. In the baseline case, the thread is already hamstrung by the low rate at which it can generate new memory references; adding bulk cache requests increases the contention for MSHRs and request ports, and confuses the cache replacement priority, worsening the short-term situation. Over longer intervals, we do see benefit from bulk-transferring the I-cache, and from transferring both L1 caches together.

Moving the D-cache by itself has almost no positive effect. This is a recurring theme in our results: the I-cache is more critical to post-migration performance, if both caches start empty. I-cache misses progress serially, while D-cache misses can often be serviced in parallel with each other, making the I-stream the clear bottleneck. In the case where all three caches are bulk copied, we see performance gains at large intervals primarily because the I-cache is included, and in fact this is less effective than copying the I-cache alone. Due to the size of the L2 cache, the transfer cost is not amortized even over 1 million commits (I + D + L2 is never better than I + D).

7.2. Limits of prefetching

Our next experiment explores the potential of post-migration prefetching. Here, we rely on an oracle prefetcher which has perfect knowledge of all future L1 block accesses. It prefetches these in order, looking far enough ahead to fill each of the L1 caches halfway. This result is shown in Figure 7. We see that the potential gains are high; despite incurring the full cost of sending the summaries and transferring the data, the oracle's perfect accuracy allows it to approach the performance of free transfers – and actually doing better at 10^3 commits and beyond – while far outpacing cache copying. This demonstrates that the underlying memory system has sufficient bandwidth to support efficient thread migration; performance depends primarily on the accuracy of our working-set summaries.

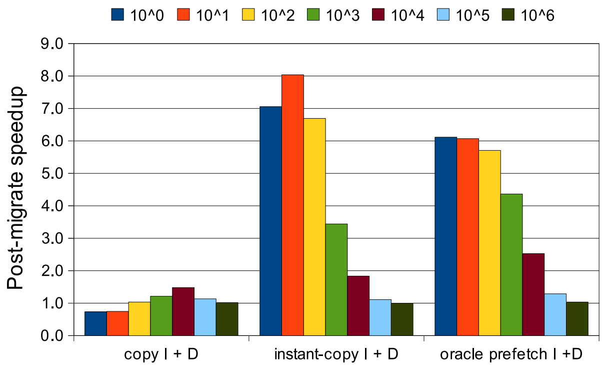

Figure 7: Impact of a future oracle prefetcher, compared to instant transfer and bulk cache copy.

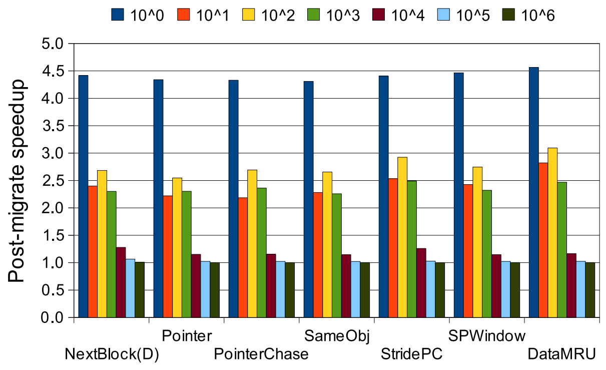

Figure 9: Impact of realistic D-stream prefetchers, combined with an oracle I-stream prefetcher.

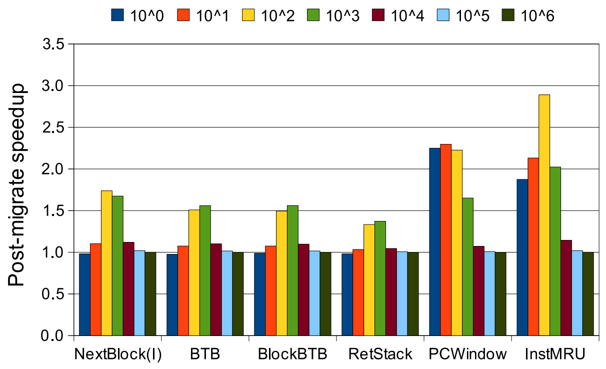

Figure 8: Impact of realistic I-stream prefetchers, combined with an oracle D-stream prefetcher.

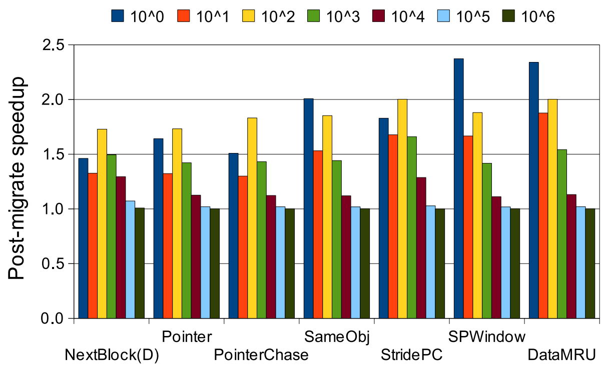

Figure 10: Impact of various combinations of realistic I- and D-stream prefetchers.

7.3. I-stream prefetching

I-stream prefetching and D-stream prefetching are synergistic; we saw this earlier, where fetching the D-stream was useless if the I-cache was empty. Therefore, to evaluate the tables of our memory logger which target the I-stream, we first assume a good solution for the D-stream. In this section, we evaluate the different I-stream tables in combination with an oracle D-stream prefetcher. The oracle still incurs overhead, but has perfect knowledge of the 1000 commits following each migration.

We see these results in Figure 8. We can't conclude much yet in terms of realistic speedup, but we do see that two very simple approaches – *PCWindow* and *InstMRU* – are quite effective over both the short and medium term. The latter has a significant advantage over moving the entire cache because it can be more timely: by requesting a subset of the cache, it moves the most relevant instructions more quickly.

7.4. D-stream prefetching

As we prefetch the I-stream into the new caches, the D-stream then becomes the bottleneck. Complementing §7.3 above, here we evaluate the tables of our memory logger which target the D-stream, temporarily assuming an oracle I-stream prefetcher with perfect knowledge of 1000 commits into the future.

Figure 9 shows these results. Here, the variations are lower than in the I-stream case, partially because of the diversity of access patterns, but also because of overlap between schemes: several of these tables track the same accesses in different ways. Again, very simple tables suffice; *DataMRU* and *StridePC* both perform well over a range of intervals, the former slightly better for shorter threads, the latter slightly better for longer threads.

7.5. Combined prefetchers

Next, we examine combinations of realistic I- and D-stream prefetchers, with no oracle knowledge. For the I-stream, we use a combination of PCWindow and InstMRU. (Results with just InstMRU were quite similar, because InstMRU partially subsumes PCWindow: the current PC is always in the MRU table.) However, by combining their behaviors, we can prefetch a larger window of instructions around the current PC than we do for the other addresses.

We combine these with several of our D-stream predictors in Figure 10. Again, we see the StridePC and DataMRU predictors each give excellent performance overall. For threads as short as 100 instructions, we achieve speedups as high as 2X using this working set prediction framework.

Table 2 shows the transfer intensity and the accuracy of several prefetching schemes: the best two prefetch com-

Prefetcher	*Blocks/migrate*	*Accuracy*
InstMRU+PCWindow+StridePC	212	64.00%
InstMRU+PCWindow+DataMRU	110	59.44%
Bulk transfer I+D	1195	48.63%

Table 2: Prefetcher activity and accuracy, mean over 200 migrations.

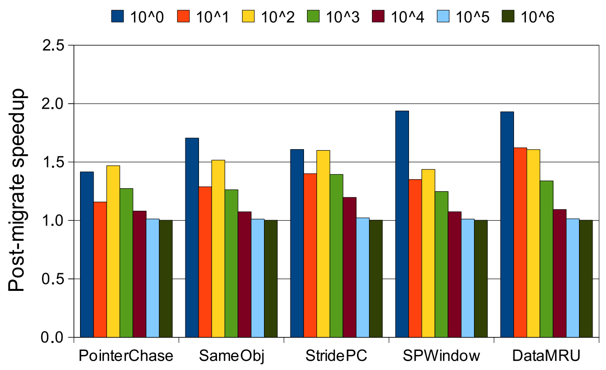

Figure 11: Realistic prefetchers, with previous-instance cache reuse and background thread movement.

binations from Figure 10, as well as bulk-copy of the L1 caches. We see from these results that our proposed migration support provides both more accurate and more directed prefetching than moving the cache state itself.

In summary, we see that a combination like InstMRU+PCWindow+DataMRU uses only two 16-entry tables, one watching the I-stream and one watching the D-stream, and enables us to as much as double the throughput of short threads.

7.6. Cross-migration data reuse

We've been migrating individual threads in an otherwise idle system, in order to evaluate prefetching absent interference from other threads. However, with no other threads, a frequently-migrating thread would quickly build up copies of its working set on each core, and gain less from prefetching. To simulate cache interference from other threads, thus far we've assumed that when a thread is migrated to a core, its entire working set has been evicted.

For a more realistic scenario, in this section we remove that restriction on cached data re-use. In order to provide cache replacement pressure on the cores which are not running the thread under evaluation, we schedule an independent workload on each (chosen randomly from SPEC2000), and move these background threads among cores periodically, outside of our prefetch-evaluation time periods. Figure 11 shows the resulting performance. Compared to Figure 10, our gains are reduced, but overall performance trends are similar. For example, mean suite-wide speedup for the *DataMRU* combination over 100 post-migrate commits has dropped from about 2.00 to 1.61. This drop is from the combination of additional cache re-use along with contention for interconnect and memory resources. However, the available performance gains are still quite high.

7.7. Impact on other threads

The experiments in §7.6 also allow us to evaluate the impact of these short bursts of prefetches on unrelated threads running on other cores. In the short term, the impact is measurable – 4% slowdown of other threads during the time the migrated thread executes its first 100 instructions – but is dwarfed by the gains of the migrated thread. The slowdown tapers off quickly, to 0.1% slowdown at 10,000 instructions.

7.8. Adding a shared last-level cache

To demonstrate that our gains are largely insensitive to memory latencies beyond the core-to-core transfers that dominate during migration, we model the addition of a shared L3 cache. We add an 8MB 16-way associative L3, with an overall load-use latency of 40 cycles (L2 latency remains 14 cycles). With this L3 added to both the baseline and experimental cases, we find that performance is nearly identical; compared to e.g. the "StridePC" experiments described for Figure 10, adding the L3 to both the baseline and the system with working set prediction, decreases the achieved speedup by a mean of 0.1% for 100 post-migrate commits, and mean 1.0% across all time scales. (In §8 we also show that overall benefits are insensitive to large changes in main-memory latency itself.)

7.9. Simple hardware prefetchers

All of our results thus far use a baseline with a fairly aggressive hardware stream-buffer based prefetcher. This prefetcher is rendered relatively ineffective during post-migration startup because it needs to learn stride patterns. It could be argued that in a system that expects frequent short threads, a simpler hardware prefetcher that ramped up more quickly might be more appropriate and could render our techniques less necessary. In fact, we find that not to be true.

For these results, we add next-block prefetchers to both L1 caches: these prefetch the successor to each block, at the first touch after that block is filled. Using these prefetchers instead of our proposed migration-targeting prefetchers, we observed only 1.010 mean speedup over 100 post-migrate commits. Over longer time scales, the benefits ramp up, to e.g. 1.104 mean speedup over 10000 post-migrate commits.

We find that over the short time scales we're most interested in for migration support, even these simple prefetchers are unable to contribute: they still fall prey to the nascent thread's slow progress, which prevents them being trained quickly enough to help, because they are being trained by demand misses. In contrast, our prefetchers are already trained at the time of migration.

8. WSM for Speculative Multithreading

We've demonstrated our techniques to be effective at mitigating the loss of cache state during thread migration in a framework that forces frequent migrations for the sole purpose of measurement. To demonstrate a practical application of working-set migration, we evaluate its ability to improve Speculative Multithreading.

In Speculative Multithreading (SpMT), loss of cache state impedes performance as execution migrates across cores [12, 13, 26]. This is a well-documented and long-standing problem. Execution that would ordinarily reside on a single core is now spread across several, creating misses and invalidate traffic where the original code experienced hits in a single cache. Additionally, coherence-based speculative multithreading requires certain data to be invalidated in caches for both squashed and committed threads, exacerbating the problem.

The SpMT scenario is different from our generic migration experiments of previous sections in three important ways. Thread start points are more deterministic (we frequently start threads at the same PC), we have less interference from other applications (we assume an idle core is not made available to other applications while waiting to spawn a new SpMT thread), and because spawned threads represent future execution, the memory loggers will experience a gap in memory addresses between the spawn point and the execution of the spawned thread. The first two differences mean that I-stream prefetching is less critical, since I-caches warm to the code of recurring threads and stay warm. Addressing the third difference is the subject of future work; however, despite all these challenges, we still see significant gains from working-set prediction on SpMT architectures.

We obtained the Speculative Multithreading framework used to evaluate a number of Hardware Transactional Memory designs [26] and modified it to include our working set migration techniques. In this framework, loops and function calls are identified for speculative parallelization entirely in hardware. Register dependencies between threads are predicted by a live-in predictor and the values are predicted using an increment predictor [21]. Memory dependencies are addressed via a modified form of Hardware Transactional Memory, specifically the OFWI design recommended by [26]. This memory design is aware of thread ordering, forwards values between threads, and detects conflicts at word granularity. Since the protocol creates invalidations, write-sharing can cause additional cache misses. We perform working-set-prediction prefetches non-transactionally, so they do not impact the read/write sets of transactions.

We evaluate the full SPEC 2000 benchmark suite with reference inputs. Each benchmark is executed using 100M-instruction simulations based on SimPoint [30]. We model dual-core execution using architectural parameters similar to [26]; these parameters include a shared L2 cache, which decreases the penalty for transferring data between cores. A lower cost of cache-to-cache transfers also lessens the criticality of migrating the working-set. Nevertheless, working set migration significantly boosts performance.

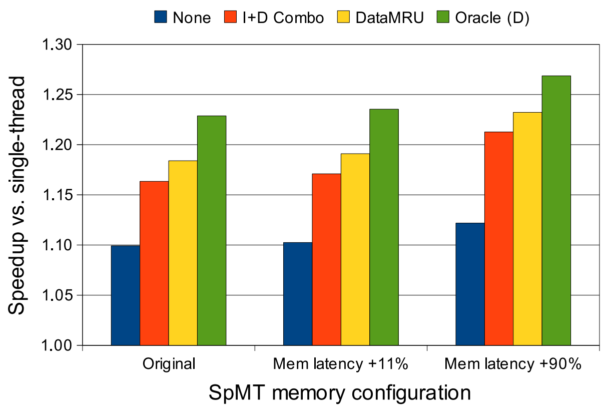

Figure 12: SPECint SpMT mean speedup across migration techniques and memory configurations. "I+D Combo" uses InstMRU, PCWindow, and DataMRU.

Speculative threads (both committed and squashed) were spawned, on average, approximately every 275 instructions. Averaged across our benchmarks, speculative threads committed 57–67 instructions before committing, depending on the working set migration support. Our migration techniques, specifically DataMRU, resulted in the most committed instructions per thread, contributing to the improved performance.

Figure 12 explores the effectiveness of our WSM techniques on the integer subset (SPECint). Focusing on the leftmost group, we see that baseline SpMT execution achieves an average speedup of 1.10, and D-stream WSM nearly doubles the overall effectiveness of speculative threading, increasing the gain from 10% to 18%. Across the entire SPEC benchmark suite, WSM increases overall gain from 24% to 32% (not pictured). When comparing the combination of prefetching both I- and D-streams ("I+D Combo") against that of data alone ("DataMRU"), we see that I-stream prefetching slightly degrades performance, since the I-caches are already warm. Although instruction prefetches are likely to be hits, they can impede the thread because they occupy cache request ports.

Figure 12 also shows results for an oracle D-prefetcher, which, at each spawn point, prefetches the memory blocks used by the next 100 instructions (the average speculative thread length is 59). This oracle achieves a 1.23 speedup on SPECint. Since this oracle prefetcher should almost entirely solve the cache locality problem for SpMT, we find that our realistic working-set prefetcher achieves most of the gains (1.18 speedup) available.

SpMT speedup is significantly hampered by the increase in average memory access time (AMAT) that occurs when we spread the computation among multiple cores. However, data working set prediction significantly mitigates the

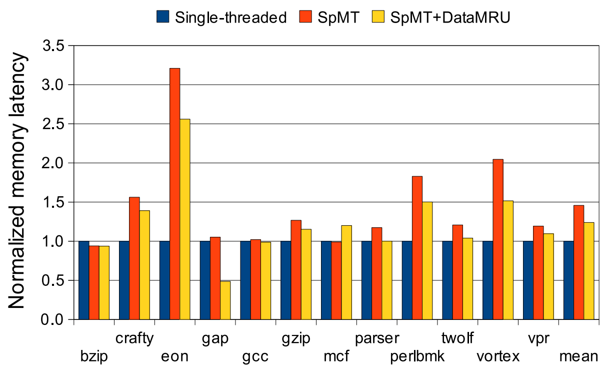

Figure 13: SPECint per benchmark average memory delay slowdown from SpMT and from SpMT with DataMRU.

AMAT inflation, as shown in Figure 13. In some cases, it reduces AMAT to near, or even below, the single-thread AMAT. In nearly all other cases, prefetching significantly reduces AMAT. As a result, overall SpMT performance improves significantly.

The one anomalous result is *mcf*. The performance of *mcf* is completely dominated by a small number of hard-to-predict “delinquent” loads [11]. SpMT benefits *mcf* less from the successful completion of spawned threads, as from the prefetching provided by those threads, which bring in hard-to-predict data. Hence, for *mcf*, speculative threads already succeed at performing critical prefetches; those critical prefetches are delayed by the extra traffic generated by DataMRU prefetching.

As mentioned in §6.1, post-migrate performance is dominated by cache-to-cache transfers. Prior work [33] shows that memory simulation models such as ours can underestimate latency by up to 25%. Since main memory access patterns are largely unaffected by WSM, the details of that simulation should not impact our relative speedups. In Figure 12 the center and right groups show the performance of SpMT and WSM with DRAM access latencies increased by 11% and 90%, respectively. The overall trends in this section are shown to be insensitive to large increases in memory latency.

To model a more aggressive SpMT system, we also evaluate the SpMT framework using a perfect register value predictor to determine if our results remain consistent without thread squashes caused by register mispredictions. Again, we see significant gains from the DataMRU working set predictor, which improves SPECint SpMT speedup from 1.24 to 1.34.

9. Summary and Conclusions

In this paper we describe a working set predictor which greatly speeds up post-migration execution. As we proceed further into the multicore era, migrations – loosely defined as scenarios where the state of a thread on one core needs to migrate to another core – will occur with greater frequency. Accelerating migration will make many proposed execution models more effective, and will make finding exploitable pockets of parallelism easier.

In this work we show that it’s critical to address I-stream performance, post-migration; otherwise, the I-cache is left to be filled by serial demand-misses. However, we improve the delivery of both instructions and data to boost the performance of short threads. We also show that simply copying cache contents is extremely ineffective over the short term: it moves too much data, at too much expense, and much of that data is not useful over the short term. We demonstrate techniques that as much as double the performance for short threads. We also demonstrate these techniques are successful at significantly improving the effectiveness of speculative multithreading. These solutions require only small, simple tables to monitor the access streams of a running thread on each core.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful insights. This research was supported in part by NSF grant CCF-1018356 and Semiconductor Research Corporation Grant 2005-HJ-1313.

References

[1] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl’s Law through EPI throttling. In *32nd International Symposium on Computer Architecture*, pages 298–309, June 2005.

[2] M. Annavaram, J. M. Patel, and E. S. Davidson. Data prefetching by dependence graph precomputation. In *28th International Symposium on Computer Architecture*, pages 52–61, July 2001.

[3] J. A. Brown and D. M. Tullsen. The shared-thread multiprocessor. In *21st International Conference on Supercomputing*, pages 73–82, June 2008.

[4] J. A. Brown, H. Wang, G. Chrysos, P. H. Wang, and J. P. Shen. Speculative precomputation on chip multiprocessors. In *6th Workshop on Multithreaded Execution, Architecture and Compilation*, pages 35–42, Nov. 2002.

[5] K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation spreading: Employing hardware migration to specialize CMP cores on-the-fly. In *12th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 283–292, Oct. 2006.

[6] P. Chaparro, J. González, and A. González. Thermal-aware clustered microarchitectures. In *22nd IEEE International Conference on Computer Design*, pages 48–53, Oct. 2004.

[7] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, and Y. N. Patt. Simultaneous subordinate microthreading (SSMT). In *26th International Symposium on Computer Architecture*, pages 186–195, May 1999.

[8] B. Choi, L. Porter, and D. M. Tullsen. Accurate branch prediction for short threads. In *13th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 125–134, Mar. 2008.

[9] J. Chung, H. Chafi, C. C. Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. The common case transactional behavior of multithreaded programs. In *12th International Symposium on High-Performance Computer Architecture*, pages 266–277, Feb. 2006.

[10] J. D. Collins, S. Sair, B. Calder, and D. M. Tullsen. Pointer cache assisted prefetching. In *35th International Symposium on Microarchitecture*, pages 62–73, Nov. 2002.

[11] J. D. Collins, H. Wang, D. M. Tullsen, C. J. Hughes, Y.-F. Lee, D. M. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *28th International Symposium on Computer Architecture*, pages 14–25, July 2001.

[12] S. L. Fung and J. G. Steffan. Improving cache locality for thread-level speculation. In *20th International Parallel and Distributed Processing Symposium*, Apr. 2006.

[13] S. Gopal, T. Vijaykumar, J. Smith, and G. Sohi. Speculative versioning cache. In *4th International Symposium on High-Performance Computer Architecture*, Feb. 1998.

[14] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *8th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 58–69, Oct. 1998.

[15] Intel. First the tick, now the tock: Next generation Intel microarchitecture (Nehalem). *Intel white paper*, 2008.

[16] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *17th International Symposium on Computer Architecture*, pages 364–373, June 1990.

[17] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. Software data spreading: Leveraging distributed caches to improve single thread performance. In *ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, June 2010.

[18] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *36th International Symposium on Microarchitecture*, pages 81–92, Dec. 2003.

[19] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *31st International Symposium on Computer Architecture*, pages 64–75, June 2004.

[20] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *28th International Symposium on Computer Architecture*, pages 144–154, July 2001.

[21] P. Marcuello and A. González. Thread-spawning schemes for speculative multithreading. In *8th International Symposium on High-Performance Computer Architecture*, pages 55–64, Feb. 2002.

[22] J. C. Mogul, J. Mudigonda, N. Binkert, P. Ranganathan, and V. Talwar. Using asymmetric single-ISA CMPs to save energy on operating systems. *IEEE Micro*, 28(3):26–41, May 2008.

[23] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *9th International Symposium on High-Performance Computer Architecture*, pages 129–140, Feb. 2003.

[24] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *21st International Symposium on Computer Architecture*, pages 24–33, Apr. 1994.

[25] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *11th International Symposium on Computer Architecture*, pages 348–354, June 1984.

[26] L. Porter, B. Choi, and D. M. Tullsen. Mapping out a path from hardware transactional memory to speculative multithreading. In *18th International Conference on Parallel Architectures and Compilation Techniques*, pages 313–324, Sept. 2009.

[27] C. G. Quiñones, C. Madriles, F. J. Sánchez, P. Marcuello, A. Gonzáles, and D. M. Tullsen. Mitosis compiler: An infrastructure for speculative threading sed on precomputation slices. In *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 269–279, June 2005.

[28] A. Roth, A. Moshovos, and G. S. Sohi. Dependence based prefetching for linked data structures. In *8th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 115–126, Oct. 1998.

[29] S. Sair, T. Sherwood, and B. Calder. Quantifying load stream behavior. In *8th International Symposium on High-Performance Computer Architecture*, pages 197–208, Feb. 2002.

[30] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 45–57, Oct. 2002.

[31] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *33rd International Symposium on Microarchitecture*, pages 42–53, Dec. 2000.

[32] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *22nd International Symposium on Computer Architecture*, pages 414–425, June 1995.

[33] S. Srinivasan, L. Zhao, B. Ganesh, B. Jacob, M. Espig, and R. Iyer. CMP memory modeling: How much does accuracy matter? In *5th Workshop on Modeling, Benchmarking and Simulation*, pages 24–33, June 2009.

[34] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *4th International Symposium on High-Performance Computer Architecture*, pages 2–13, Jan. 1998.

[35] R. D. Strong, J. Mudigonda, J. C. Mogul, N. L. Binkert, and D. M. Tullsen. Fast switching of threads between cores. *ACM SIGOPS Operating Systems Review*, 43(2):35–45, Apr. 2009.

[36] M. A. Suleman, O. Mutlu, J. A. Joao, Khubaib, and Y. N. Patt. Data marshaling for multi-core architectures. In *37th International Symposium on Computer Architecture*, pages 441–450, June 2010.

[37] D. M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd International Computer Measurement Group Conference*, pages 819–828, Dec. 1996.

[38] E. Tune, R. Kumar, D. M. Tullsen, and B. Calder. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. In *37th International Symposium on Microarchitecture*, pages 183–194, Dec. 2004.

[39] C. B. Zilles and G. S. Sohi. Understanding the backward slices of performance degrading instructions. In *27th International Symposium on Computer Architecture*, pages 172–181, June 2000.

[40] C. B. Zilles and G. S. Sohi. Execution-based prediction using speculative slices. In *28th International Symposium on Computer Architecture*, pages 2–13, June 2001.