SLICC: Self-Assembly of Instruction Cache Collectives for OLTP Workloads

Islam Atta[†] Pınar Tözün[‡] Anastasia Ailamaki[‡] Andreas Moshovos[†]

[‡]École Polytechnique Fédérale de Lausanne
{pinar.tozun, anastasia.ailamaki}@epfl.ch

[†]University of Toronto
{iatta, moshovos}@eecg.toronto.edu

Abstract

Online transaction processing (OLTP) is at the core of many data center applications. OLTP workloads are known to have large instruction footprints that foil existing L1 instruction caches resulting in poor overall performance. Prefetching can reduce the impact of such instruction cache miss stalls; however, state-of-the-art solutions require large dedicated hardware tables on the order of 40KB in size.

SLICC is a programmer transparent, low cost technique to minimize instruction cache misses when executing OLTP workloads. SLICC migrates threads, spreading their instruction footprint over several L1 caches. It exploits repetition within and across transactions, where a transaction's first iteration prefetches the instructions for subsequent iterations or similar subsequent transactions. SLICC reduces instruction misses by 58% on average for TPC-C and TPC-E, thereby improving performance by 68%. When compared to a state-of-the-art prefetcher, and notwithstanding the increased storage overheads (42× as compared to SLICC), performance using SLICC is 21% higher for TPC-E and within 2% for TPC-C.

1. Introduction

Online transaction processing (OLTP) is a multi-billion dollar industry that increases 10% annually [7]. OLTP needs have been driving innovations by both database management system and hardware vendors, and OLTP performance has been a major metric of comparison across vendors [11, 31, 32]. Unfortunately, modern cloud and server infrastructures are not tailored well for the characteristics of OLTP applications [4]. Literature shows that OLTP workloads are memory bound; memory access stalls account for 80% of execution time, most of which are due to first-level instruction cache misses [15, 4, 28]. Software [9] and hardware [14, 26, 3, 5] efforts are trying to alleviate stall time related to instruction misses.

Transactions of canonical OLTP systems are randomly assigned to worker threads, each of which usually runs on one core of a modern multi-core system. The instruction footprint of a typical transaction does not fit into a single L1-I cache, thus thrashing the cache and incurring a high instruction miss rate. Although L2 and L3 caches are growing in size, today's technology and CPU clock cycle constraints prevent deploying L1-I caches larger than 32KB. As this work demonstrates, the instruction footprint of a typical OLTP transaction fits comfortably in the aggregate L1-I cache capacity of modern many-core chips. Provided that there is sufficient code reuse, spreading the footprint of transactions over multiple L1-I caches would reduce instruction cache misses. Fortunately, as corroborated by our experimental results, OLTP workloads exhibit a high-degree of instruction reuse both within a transaction and across concurrently running transactions [3, 9].

This paper proposes *SLICC* (Self-Assembly of Instruction Cache Collectives), a hardware technique that utilizes thread migration to minimize instruction misses for OLTP workloads. SLICC divides the instruction footprint of a transaction into smaller code segments and spreads them over multiple cores, so that each L1-I cache holds part of the instruction footprint. As part of this process the L1-I caches self-assemble to form a *collective* that reduces the instruction misses for this transaction and other similar ones. SLICC exploits intra- and inter-thread instruction locality in two orthogonal ways: (1) A thread looping over multiple code segments spread over multiple caches observes a lower miss rate (as opposed to a conventional system in which each segment would evict the others from the cache), thereby avoiding thrashing. (2) A preamble thread effectively prefetches and distributes common code segments for subsequent threads, thereby reducing the total miss rate. As execution progresses, old cache collectives are naturally disassembled and new ones are formed to hold the footprints of new transactions.

As opposed to previous OLTP instruction miss reduction techniques, SLICC is a hardware solution, that avoids undesirable instrumentation, utilizes available core and cache capacity resources, covers user as well as system-level code, and requires no changes to the existing user code or software system. SLICC incurs overheads due to thread migration; thus, context switching and increases in data misses must be amortized to improve performance. A hardware thread migration mechanism provides a programmer transparent solution that has low context switching overheads, and the positive impact of the reduction in instruction misses outpaces the extra data misses.

To evaluate SLICC, we execute two popular transactional benchmarks, TPC-C [29] and TPC-E [30], as well as a MapReduce [4] cloud workload. Our experiments show that, on average, thread migration eliminates 56% of the L1 instruction misses resulting in a 68% overall performance improvement over the baseline (described in Section 5.1). Compared to PIF [5], a state-of-the-art instruction prefetcher, SLICC improves performance by 21% for TPC-E and comes within 2% for TPC-C, with only 2.4% of relative storage area overhead. SLICC is also robust as it does not affect the performance of MapReduce [4], a cloud workload, which has a relatively small instruction footprint. In summary, this paper makes the following contributions:

- It characterizes the memory behavior of TPC-C [29] and TPC-E [30] showing that transactions suffer from instruction misses, and that their instruction streams exhibit intra- and inter-transaction recurring patterns leading to eviction of useful blocks that are re-accessed (96% of capacity misses are for instructions).
- It demonstrates that recently proposed cache replacement policies [24, 12] reduce instruction misses by 8% on average for the best policy, but leave ample room for improvement.
- It presents SLICC, a hardware thread migration algorithm, and shows that it reduces instruction misses by 56% on average with an overall 68% performance improvement for OLTP.

The remaining of this document is organized as follows. Section 2 analyzes the nature of the problem and Section 3 sets the requirements for an ideal solution. Section 4 describes the automated thread

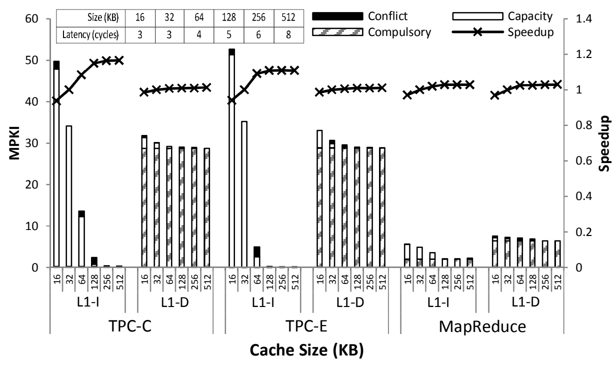

Figure 1: Instruction and data L1 misses and relative performance as a function of cache size.

migration algorithm SLICC. Section 5 demonstrates experimentally the performance benefits of SLICC. Sections 6 reviews related work, while Section 7 presents our conclusions.

2. Instruction MISS Analysis

This section examines the OLTP memory behavior that motivates thread migration for instruction cache miss reduction. The analysis targets TPC-C [29] and TPC-E [30], which resemble state-of-the-art commercial OLTP applications. We contrast their behavior with MapReduce [4], a data center workload which has a smaller instruction footprint. We find that for OLTP transactions:

1. Most instruction misses are due to limited cache capacity, whereas most data misses are compulsory.
2. The instruction footprint of most transactions would fit in the aggregate L1 instruction cache capacity of even small scale chip multiprocessors (eight cores). The same is not true for data footprints.
3. Existing non-LRU cache replacement policies reduce the instruction miss rate, but only by a fraction of what would be possible with larger caches.
4. There is intra-thread locality, but over code regions that are larger than a typical L1 cache size.
5. There is significant inter-thread locality, particularly across threads of the same transaction type.

2.1. OLTP Instructions and Data Misses

In typical multi-threaded OLTP systems, a transaction is assigned to a worker thread. Thus multiple similar transactions (threads) usually run concurrently. Individual threads, whose memory footprints do not fit in the L1 cache, suffer from high miss rates. Ferdman *et al.* show that in OLTP, memory stalls account for up to 80% of the execution time [4], while Tözün *et al.* show that instruction stalls account for 70–85% of the overall stall cycles [28].

2.1.1. L1 Miss Breakdown We further analyze instruction and data L1 misses. Figure 1 shows the number of misses per kilo-instructions (MPKI) for a range of L1 instruction (L1-I) and data (L1-D) cache sizes. Section 5 details the experimental methodology. We first vary the L1-I cache size (16KB–512KB) while keeping the L1-D cache size at 32KB (our baseline), and then we vary the L1-D cache size while keeping the L1-I at 32KB. CACTI 6 [20] is used to model the access latencies of different cache sizes. Figure 1 shows a breakdown of instruction and data L1 misses into three categories: capacity, conflict and compulsory [10]. By identifying where most misses come from, we highlight the reasons behind the memory stalls; is it cache size, associativity, or cold misses?

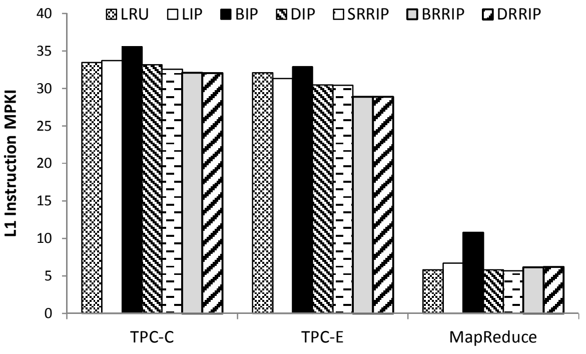

Figure 2: MPKI with different cache replacement policies.

Figure 1 shows that for OLTP workloads, capacity misses dominate instruction misses. This implies that the instruction footprint does not fit in the cache and has lots of reuse; cache blocks are evicted from the cache before they are re-referenced. Hence larger L1-I caches, which can hold cache blocks for longer periods, can reduce instruction misses. To keep up with the CPU clock speeds, technology constraints have limited the sizes of L1 instruction caches to about 32KB today. With 32KB caches, instruction capacity misses are an order of magnitude more than data capacity misses. Compulsory misses, which occur for the first reference to each unique cache block, dominate data misses. Thus, larger data caches can do little to reduce data misses. For 32KB caches, compulsory data misses are an order of magnitude more than compulsory instruction misses. Unless data is prefetched, data misses cannot be reduced.

MapReduce is a cloud workload featuring a relatively smaller instruction footprint [4]. Since 71% of the total L1 misses are compulsory for 32KB caches, larger L1 instructions or data caches are not as beneficial.

Figure 1 also shows overall performance improvement normalized to the 32KB baseline. Performance improvements with larger L1-D caches are negligible at 1%, but can be as high as 16% with larger L1-I caches (MapReduce shows less than 3% improvement). If increasing cache size did not also increase latency, performance improvements would be higher; for example, a 512KB L1-I with the latency of a 32KB L1-I would result in a 61% performance improvement for TPC-C.

We conclude that (a) OLTP transactions have instruction footprints that, while larger than typical L1-I caches, could fit in the aggregate L1-I capacity of modern multi-cores. (b) OLTP data footprints are much larger and cannot fit onto the aggregate L1-D capacity of modern multi-cores. Additionally, (c) OLTP instruction streams exhibit a significant reuse over regions that exceed typical L1-I cache sizes leading to the eviction of useful blocks that are re-accessed.

2.1.2. Replacement Policies Conventional caches often use some approximation of LRU replacement. Qureshi *et al.* show that some workloads, including those that have long-term reuse, are not LRU-friendly [24]. For such workloads, LRU cycles through a large footprint while it would be best to keep at least some part of the footprint cache resident. They modify LRU by introducing new static (LIP, BIP) and dynamic (DIP) insertion policies where newly accessed blocks are not necessarily inserted at the most-recently-used position of the LRU stack. Jaleel *et al.* propose the SRRIP, BRRIP, and DRRIP re-reference interval based insertion policies [12]. Their Re-reference Insertion Prediction (RRIP) chain represents the order in which blocks are predicted to be re-referenced. The block at the

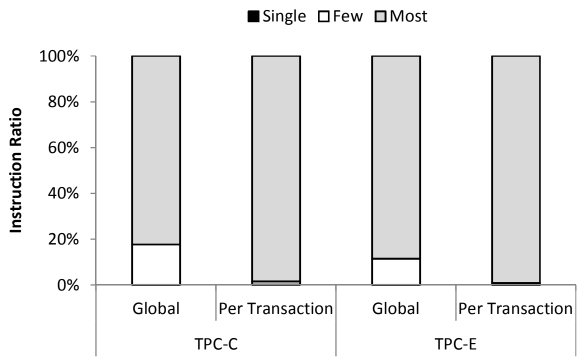

Figure 3: Breakdown of accesses accordingly to instruction block reuse.

head of the RRIP chain is predicted to have a *near-immediate* re-reference interval, while the block at the tail of the RRIP chain is predicted to have a *distant* re-reference interval. On a cache miss, a *distant* block is replaced. Re-references to a block promote its position towards the head of the chain.

Figure 2 reports the MPKI for these replacement policies for the baseline 32KB L1-I cache. BRRIP and DRRIP perform best reducing misses by an average of 8% over LRU. This reduction is only a fraction of what is possible with larger caches as Figure 1 showed.

Thrashing applications favor Bimodal RRIP (BRRIP), which predicts a *distant* re-reference interval for most blocks [12]. Dynamic RRIP (DRRIP) selects the best policy from Static RRIP (SRRIP) and BRRIP at runtime. Figure 2 shows that DRRIP chose BRRIP most of the time. Contrary to the Least Insertion Policy (LIP), which promotes a referenced block to the MRU position, BRRIP uses access frequency to promote cache blocks gradually towards the head of the chain.

Thus, we see that the recurring patterns exhibited by OLTP instruction streams have relatively long periods that cannot be fully captured by existing insertion/replacement policies. Nevertheless, replacement policies are orthogonal to thread migration.

2.1.3. Redundancy Across Threads Chakraborty *et al.* profile OLTP instruction accesses and report an 80% redundancy across multiple cores for user and OS code [3]. Figure 3 corroborates these results and further shows that 98% of the instruction cache blocks are common among threads executing the same transaction type; although similar transactions do not follow the exact same control flow path, they have common code segments at a coarser granularity. Figure 3 shows a breakdown of all instruction cache accesses classified according to the reuse experienced by the accessed block over the duration of the application. The figure presents three coarse reuse categories: *single*, *few* and *most* that correspond to blocks accessed by only one thread, at most, or more than 60% of all the threads, respectively. This behavior highlights an opportunity to reduce instruction misses by exploiting temporal locality across multiple threads, particularly threads of the same transaction type.

3. Thread Migration for OLTP

SLICC exploits the aggregate L1-I cache capacity of many cores and the availability of multiple concurrent threads with inherent code commonality. It is a hardware transaction scheduling algorithm that spreads the instruction code footprint across multiple L1-I caches. SLICC dynamically pipelines and migrates threads to cores that are predicted to hold the code blocks to be accessed next. By migrating

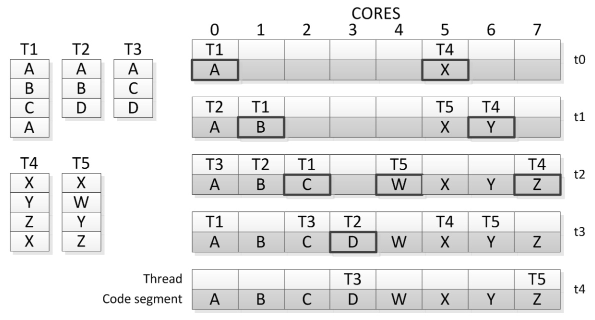

Figure 4: Thread migration common code segment reuse example. Left: T1-3 and T4-5 are threads running similar transactions. Right: 8-core system. The shaded area is the cache activity (thick border = warm-up phase).

threads, SLICC virtually increases the cache capacity observed by a thread, and thus, the reuse of instruction blocks brought in to the cache, avoiding thrashing.

The observations of Section 2.1 support SLICC's approach: (a) it can avoid many instruction misses by virtually increasing the L1-I cache size per thread; and (b) it can effectively increase locality by grouping similar transactions together. SLICC tends to increase intra- and inter-thread instruction locality.

3.1. Example Scenario

Figure 4 exemplifies how thread migration can reduce instruction misses. Threads T1-T5 are scheduled to run on an 8-core system, where T1-T3 and T4-T5 execute respectively transactions of the same type. The transactions' footprints are divided into code segments, where each *segment* fits in the L1-I cache of a single core, but two segments would not fit together. T1 executes the following code segments in order: A-B-C-A. Thus, its instruction footprint is 3× larger than the L1-I cache size. Since T2 and T3 are of the same type as T1, they share common segments with T1, but their execution paths are not identical. A conventional system would schedule T1, T2, and T3 on separate cores, and since their footprints are larger than the L1-I cache size, each thread would suffer all instruction misses.

Figure 4 (right) demonstrates an ideal scenario for thread migration. Initially (at time t0), T1 runs on core-0. When it is done with code segment A, and so all cache blocks for A have been brought in to the cache, T1 migrates to core-1 (at t1), where it continues execution fetching cache blocks of segment B. At the same time, T2 can be scheduled to start execution on core-0 (at t1), ideally reusing all blocks in A (miss rate close to zero). We refer to this as inter-thread reuse. The process continues and T1 warms-up caches 1 and 2 with B and C, respectively. At t3, when T1 goes back to A, it migrates to core-0 benefiting from intra-thread reuse.

Migration is beneficial even if T1-T3 do not follow identical paths, as segment D illustrates. Since T1 did not touch segment D, T2 will suffer due to the corresponding misses while executing on core-3. If T3 follows suit on core-3, it will not suffer any instruction misses for segment D.

T4-T5 that access different code segments benefit as well if they get assigned to a different set of cores, avoiding conflicts with T1-T3. This process applies to all subsequent threads: if they touch a code segment that exists in some L1-I cache, they migrate to the corresponding core and avoid missing for these segments.

Without thread migration multiple requests would be repeatedly sent out for the same cache block from multiple cores. With thread migration, an instruction cache block is, under the ideal scenario, requested only once, and reused multiple times.

3.2. SLICC Requirements

Based on our preceding discussion we identify three requirements for SLICC. SLICC should dynamically detect: (a) *When* a thread should migrate, i.e., when is the current cache *full*; and (b) *where* the thread should migrate to, i.e., which remote cache, if any, holds the code segment the thread will touch next. Since transactions vary in their control flow, SLICC should (c) not impose any specific pipelining, i.e., it should not restrict similar threads to follow the exact same path.

In order to meet these requirements, SLICC needs to maintain runtime information about caches and individual threads to be able to make judicious migration decisions. First, SLICC needs a mechanism to determine whether the cache is filled-up with useful cache blocks or not. In addition, with respect to a given core, SLICC should be able to predict which remote core holds the cache blocks that will be touched next.

The basic SLICC design is a type-oblivious algorithm, i.e., no information is provided about which threads resemble the same transaction type. Information about thread types could enhance SLICC's process. Thus, there are several alternatives. On one extreme, the hardware can dynamically migrate threads to cores irrespective of their types. On the other extreme, the software layer can transfer knowledge about thread types. In between, threads can be pre-processed to detect threads with similar starting address ranges. We detail the three alternatives in Section 4.

3.3. Effect on Data Misses

When a thread migrates, it leaves data that might be reused behind. This may increase the data miss rate. Section 5 shows that while SLICC does increase the data miss rate, the benefit from reducing instruction misses outpaces the performance loss due to data miss increase.

Intuitively, depending on the workload, instruction misses can impact performance more than data misses. For example, modern architectures use instruction level parallelism (ILP) to hide data miss latencies. Instruction misses restrict instruction supply, rendering such ILP techniques less effective. Existing core architectures make no effort to balance the relative cost of the two types of misses. SLICC provides a way of balancing the relative costs of data vs. instruction misses. In Section 5 we show that SLICC reduces the overall L1 miss rate.

4. SLICC Design

SLICC exploits intra- and inter-thread locality. (1) It virtually increases the L1-I cache capacity observed by a thread; thus, it improves locality within a thread. (2) It pipelines similar threads, such that one thread fetches instruction cache blocks that are reused by many threads.

This work presents three different SLICC designs. The first design (SLICC) is transaction-type-oblivious, while the other two exploit transaction type information. Given that threads of the same transaction type tend to have similar footprints, knowing each thread's transaction type can lead to better migration decisions. The transaction type information is either provided by the software (SLICC-SW), or detected at runtime by the hardware based on the initial instruction sequence each thread executes (SLICC-Pp). These implementations represent the two extremes and an in-between solution in terms of hardware/software co-operation. Future work may look at other alternatives.

4.1. Transaction-Type-Oblivious SLICC

SLICC is a dynamic hardware thread scheduling and migration algorithm that is programmer transparent. SLICC attempts to partition on-the-fly the instruction footprint of transactions into several segments where each segment fits in the L1-I cache, but two segments do not fit together. Ideally: (1) a thread will migrate to another core when it starts touching a different segment, and (2) the destination core will already have the segment cached.

Figure 5 shows the sequence of events that lead to thread migration. In the steady state, each core has a running thread and a hardware queue of waiting threads. Using a naïve load-balancing strategy, newly arrived threads are scheduled to the least congested core (i.e., the core with the least number of waiting threads). A SLICC agent at each core continuously monitors execution locally in order to determine whether (Q.1) the local cache is filled-up with useful instruction blocks, if so, (Q.2) whether these blocks are useful to the current thread and for how long, and (Q.3) where to migrate to if needed.

(Q.1) Is the cache full with useful blocks? As a thread starts executing on a core it may experience many misses. If the cache contains a segment that may be useful for other threads, it is best to migrate the current thread to another core. Otherwise, it is best to allow the current thread to load a new segment in the cache. SLICC uses a "cache full" detection heuristic to make this decision. Initially, all caches are "empty". To detect whether a cache has been filled up with a *segment*, SLICC counts the number of misses using a resettable, saturating miss counter (*MC*) local to each core. When the number of misses exceeds the threshold, *fill-up_t*, the cache is considered *full*. In the long run, all MCs will saturate, preventing new segments from being cached effectively due to premature thread migration. To create opportunities for loading new segments, SLICC resets the MC when the core's thread queue becomes empty. The currently cached blocks are not flushed, so if a subsequent thread requires the same segment it will still find it there. However, a thread touching a new segment will be given the opportunity to cache it.

(Q.2) Are the current cache contents useful to this thread and for how long? When running a thread on a *full* cache, SLICC tries to determine whether the thread is going over the cached segment, or whether it is about to move to a new segment. For this purpose SLICC measures *miss dilution*, that is, the recent frequency of misses (detailed in Section 4.2.2). If miss dilution is low, then SLICC predicts that thread is only temporarily diverting away from the cached segment. Since the thread will converge again soon, it is best to not migrate to benefit from the forthcoming instruction reuse. If miss dilution is high, then SLICC predicts that the thread is moving to a different segment. If it continues execution on this core it will evict useful cache blocks, which could be reused by other threads. SLICC predicts that it might be better to migrate the thread elsewhere. The question at this point becomes *where* to go?

(Q.3) Where to migrate to? Ideally, SLICC would migrate a thread to a cache that has the thread's next segment. SLICC attempts the following in order: (1) If the thread is going to touch a code segment that is available on another core, the thread migrates there.

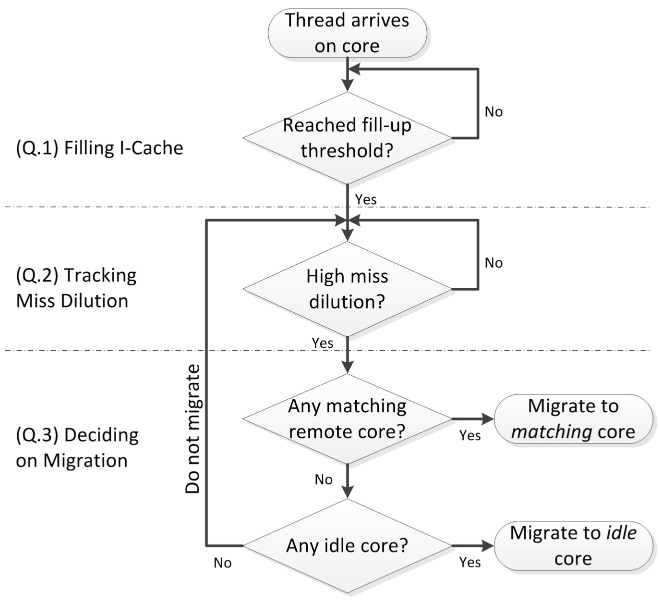
Thread arrives on core
Reached fill-up threshold?
No
(Q.1) Filling I-Cache
Yes
High miss dilution?
No
(Q.2) Tracking Miss Dilution
Yes
Do not migrate
Any matching remote core?
Yes
Migrate to *matching* core
(Q.3) Deciding on Migration
No
Any idle core?
Yes
Migrate to *idle* core
No

Figure 5: Thread Migration Algorithm.

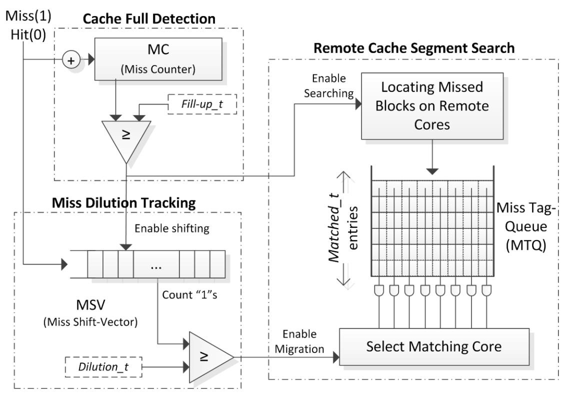

Figure 6: SLICC Architecture.

(2) Otherwise, the thread migrates to an idle core, if any. (3) The thread stays put. In the last case, migrating the thread would incur overheads and would evict remotely cached segments that may be useful for other threads. SLICC opts for incurring the instruction misses locally avoiding the migration overhead.

To detect which, if any, remote cache has the next segment, SLICC uses a short sequence of *matched_t* number of tags of recent misses, predicting that they form the preamble of the next segment. Conceptually, once SLICC decides to try to migrate a thread, it searches all remote L1-I caches for these recently missed tags. Section 4.2.3 explains how this search can be implemented including an incremental method that uses the existing coherence protocol responses.

Figure 5 summarizes the execution stages of a thread on a core until it migrates, or completes execution.

4.2. Implementation Requirements

Figure 6 shows that SLICC's implementation comprises: (a) a cache full detector, (b) a miss dilution tracker, and (c) a remote cache segment search unit. SLICC uses hardware thread migration, and thus, interacts with the OS as Section 4.4 explains in more detail. The three aforementioned units, described subsequently, track all cache accesses, including speculative ones.

4.2.1. Cache Full Detection A $log_2(L1I\ cache\ blocks)$ wide saturating miss counter (*MC*) continuously counts the number of misses. When MC saturates at a value of *fill-up_t* SLICC assumes that the cache has now captured a full segment and may trigger migrations accordingly. We experimentally found that using a value in the order of $\frac{cache\ size}{2}$ for the *fill-up_t* threshold works reasonably well, with little sensitivity to the exact value of this parameter. Other fill-up detection mechanisms may be possible but are beyond the scope of this paper.

4.2.2. Miss Dilution Tracking It is not always beneficial to migrate threads immediately after a cache becomes full or when a thread incurs a few misses. SLICC must predict whether the thread is only temporarily diverging due to conditional control flow or whether it is moving to a completely different segment. Furthermore, since threads have to miss for a few blocks before migrating (*matched_t* tags must be located on a remote cache), a few useful cache blocks may be evicted, creating gaps in the exiting segment and causing a corresponding number of misses for subsequent threads. Finally, a thread may immediately loop back to the same code segment or may temporarily follow a somewhat different path after being selected for migration.

SLICC handles these cases by considering the frequency of instruction misses; it restricts migration to the cases when a thread starts to miss more frequently. If the thread is moving to a new segment, it will incur more misses than hits. SLICC counts the number of misses in a window of recent accesses. When this count is above the dilution threshold, *dilution_t*, migration is enabled. The miss shift-vector (*MSV*) is a 100-bit FIFO shift vector recording the hit/miss history for the last 100 cache accesses (enabled when cache is filled-up). A logic-0 and logic-1 represent a cache hit and miss, respectively. When the number of logic-1 bits reaches a threshold (*dilution_t*), SLICC enables migration. SLICC resets the *MSV* with every migration.

4.2.3. Remote Cache Segment Search When SLICC decides to migrate a thread it has to determine which cache, if any, contains the segment the thread is executing. To do so, SLICC records recently missed tags in the Missed Tag Queue (*MTQ*), which is a *matched_t* entry FIFO of *n*-bit entries, where *n* is the number of cores. A logic-1 on bit index *C* for MTQ entry *i* indicates that the *i*th recently missed cache block was cached at core *C*. Thus, by ANDing all bits at index *C* we know whether core *C* holds all the recently missed cache blocks. This information does not have to be exact or accurate, since it is used by a prediction mechanism. SLICC gathers this information *incrementally* as misses occur and stores it in the MTQ. The remote cache segment search is distributed and the decision is made locally by the core we migrate from. A directory coherence protocol could report the complete or partial sharing vector for misses that are tracked by the MTQ.

Alternatively, or if the coherence protocol is snoop-based, SLICC could broadcast the missed tags as they occur and explicitly request that remote cores identify themselves. On snoop coherence systems, these requests can piggyback on the existing snoop requests. Searching remote L1-I caches requires extra bandwidth on the remote caches that is proportional to the number of missed tags and cores.

To avoid this bandwidth overhead, we use an approximate cache signature in the form of a partial-address bloom filter that supports evictions [23]. When the index size of the bloom filter is larger than the cache set index, collisions occur only within sets. Hence on evictions, only the set of the evicted block is checked for collisions. Every core maintains such a filter, representing a superset of the currently cached blocks. In this design, once migration is triggered, remote-cache search requests are answered by the approximate signature, avoiding contention with the original cache references of the remote core. In Section 5.3, we evaluate the tradeoff of the bloom filter's accuracy versus its size. We find that for a 32KB cache, a 256B bloom filter is sufficient.

If no matching remote cache is found, SLICC will attempt to find an idle core. SLICC either broadcasts a request for idle cores to report, or piggy-backs this information on the responses received during the miss tag search phase. Thread migrations are relatively infrequent (every 3.2K instructions on average), reducing the relative overhead of remote cache segment and idle core searching.

4.3. Exploiting Transaction Type Information

Section 2.1.3 showed that the instruction footprint overlap is higher among threads of the same transaction type. The basic SLICC does not directly exploit this phenomenon. It tries to detect, on-the-fly, whether a thread matches the segment on the core it is currently executing. Thus, a thread of type *X* may partially kick-out cache blocks used by threads of type *Y*. If the transaction type for each thread were known, SLICC could schedule similar threads on the same set of cores to reduce conflicts. We propose two SLICC variants that exploit such thread transaction type information.

4.3.1. Assigning Transaction Types *SLICC-SW* relies on the OLTP software layer to annotate each thread upon launch with a transaction type. This guarantees correctness, but requires some modifications to the software/hardware interface.

Alternatively, *SLICC-Pp* uses a hardware preprocessing phase to assign types to threads as they launch. SLICC-Pp exploits the observation that in OLTP the first few instructions executed are the same for same-type threads, while they differ across different-type threads. SLICC-Pp only needs to know when a new thread is launched. A middle-ware layer assigns threads in groups to a core devoted for this purpose (*scout core*). There, each thread executes a few tens of instructions, while the instruction addresses are hashed. The resulting values are used as thread type identifiers. Experiments show that SLICC-Pp is 100% accurate when executing a small number of instructions. SLICC-Pp dedicates one core for pre-processing.

4.3.2. Type-Aware Migration Using thread type information, SLICC groups similar threads into teams. Creating teams is useful for two reasons: (1) it groups similar transactions to improve opportunities for co-scheduling and overlap, and (2) it helps scheduling reduce waiting times. For each thread SLICC records a unique numerical ID, a type ID, and an arrival timestamp. The timestamp of a team is that of its oldest thread. The oldest team is scheduled, without pre-emption if possible.

We intuitively design a scheduling algorithm that maximizes the core utilization and reduces the queuing delay of threads. Team sizes differ and for an *N*-core architecture we categorize them into *large* ($1.5\times$ to $2\times$ *N* threads), *medium* ($0.5\times$ to $1.5\times$ *N* threads), and *small* (less than $0.5 \times N$ threads) teams. Cores are time-multiplexed among teams. When large teams are scheduled, they are allowed to execute on all cores. Medium size teams are limited to half the resources ($0.5 \times N$ cores). Threads of a small team are treated as stray threads, and are not grouped. Rather, stray threads are scheduled, individually, to idle cores, or in parallel with a medium team. For SLICC-SW and SLICC-Pp, when a team of threads completes execution, SLICC resets all *MC*s, *MTQ*s and *MSV*s.

Table 1: Workload Parameters.

TPC-C-1	1 warehouse, 84 MB Wholesale supplier
TPC-C-10	10 warehouses, 1 GB Wholesale supplier
TPC-E	1000 customers, 20 GB Brokerage house
MapReduce	Hadoop 0.20.2, Mahout 0.4 library Wikipedia page articles (12 GB)

4.4. Support for Thread Migration

To allow for queuing threads, the thread migration performed in SLICC transfers architectural register files as in Thread Motion [25]. The thread's context is saved in the L2 cache closest to the target core and is then retrieved at the target core. This minimizes the set-up time for the thread. Since modern commercial processor technologies (e.g., Intel Virtualization (VT) [33] and AMD Secure Virtual Machine (SVM) [1]) provide hardware support for thread migration, minimal modifications are required to make the migration process transparent to higher software layers.

Canonical OS kernels are responsible for assigning threads to cores. Hardware support for thread migration that is transparent to higher layers avoids any software overhead. Otherwise, the OS scheduler must be informed about these migrations. An alternative is a hybrid system in which hardware mechanisms provide counters and migration acceleration, while leaving the policy choice to software. This enables easier integration between existing schedulers and platforms with virtualization support.

5. Evaluation

Our evaluation: (1) Studies the configuration thresholds for SLICC (Section 5.2). (2) Determines the trade-off between bloom filter size and remote cache segment search accuracy (Section 5.3). (3) Demonstrates SLICC's effect on instruction (Section 5.4) and data misses (Section 5.5), compared to the baseline. (4) Reports the performance improvement with the different flavors of SLICC compared to the baseline and to a state-of-the-art instruction prefetcher, PIF [5] (Section 5.6). (5) Estimates the HW cost for SLICC's components (Section 5.7). (6) Reports statistics about remote cache segment search activity (Section 5.8).

5.1. Methodology

Current operating systems do not support thread migration at the hardware level. The OS kernel assumes full control over thread assignment in multicore environments. To work around this limitation, we extract x86 execution traces using PIN [18], which are annotated to identify transactions. We then replay traces, modeling the timing of all events and maintaining the original thread sequence. We modify the Zesto x86 multicore architecture simulator [17]. Previous work shows that migrating threads to a set of dedicated cores to execute system level code improves performance [3]. While this work studies migration of user-level code, SLICC is generic and can apply to system level code as well. We model thread migrations by injecting writes and reads for all architectural state and thread context information.

We examine one scale-out workload and two server workloads as described in Table 1 [4]. TPC-C [29] and TPC-E [30] run on top

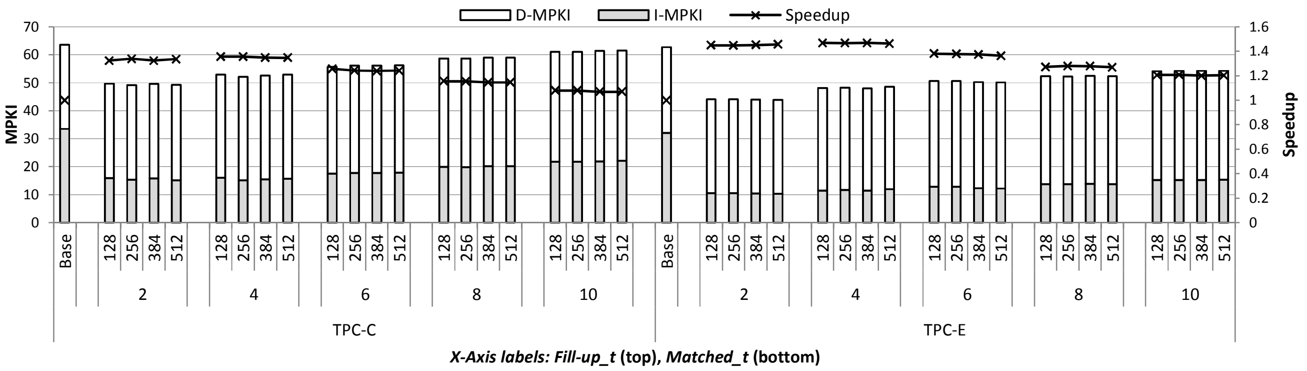

Figure 7: MPKI and relative performance as a function of *fill-up_t* and *matched_t* thresholds.

Table 2: System Parameters.

Processing Cores	16 OoO cores, 2.5GHz 6-wide Fetch/Decode/Issue 128-entry ROB, 80-entry LSQ BTAC (4-way, 512-entry) TAGE (5-tables, 512-entry, 2K-bimod)
Private L1 Caches	32KB, 64B blocks, 8-way 3-cycle load-to-use, 32 MSHRs MESI-coherence for L1-D
L2 NUCA Cache	Shared, 1MB per core, 16-way 64B blocks, 16 banks 16-cycle hit latency, 64 MSHRs
Interconnect	4×4 2D Torus, 1-cycle hop latency
Memory	DDR3 1.6GHz, 800MHz Bus, 42ns latency 2 Channels / 1 Rank / 8 Banks 8B Bus Width, Open Page Policy t_{CAS}-10, t_{RCD}-10, t_{RP}-10, t_{RAS}-35 t_{RC}47.5, t_{WR}-15, t_{WTR}-7.5 t_{RTRS}-1, t_{CCD}-4, t_{CWD}-9.5

of the scalable open-source storage manager Shore-MT [13]. The client-driver and the database are kept on the same machine, the buffer-pool is set big-enough to keep the whole database in memory, and due to the unavailability of a sufficiently fast I/O subsystem we flush the log to RAM. We simulate 1K tasks or approximately 1.1B instructions. We use two different databases in TPC-C-1 and TPC-C-10 to demonstrate that SLICC remains effective even with a larger database. TPC-C and TPC-E have larger instruction and data footprints compared to other scale-out workloads [4]. To demonstrate SLICC's robustness, we study the MapReduce CloudSuite workload [22], which does not have a large instruction footprint [4]. MapReduce divides the full input dataset across 300 threads, each performing a single map/reduce task. We focus most of our evaluation on TPC-C-1 (referred as TPC-C) and TPC-E.

Table 2 details the baseline architecture. We use misses per kilo instructions (MPKI) as our metric for instruction (I-MPKI) and data (D-MPKI) misses. We measure performance by counting the number of cycles it takes to execute all transactions. With *N*-core, our baseline architecture can run up to *N* concurrent threads with the OS making thread scheduling decisions. SLICC manages a thread pool of up to *2N* threads. Unless otherwise indicated, all SLICC results are for the SLICC-SW configuration – i.e., the SW layer transfers knowledge about thread types to the HW layer.

5.2. Exploring SLICC's Parameter Space

SLICC utilizes three thresholds to make thread migration decisions: *fill-up_t*, *matched_t* and *dilution_t*. This section explores their effect on L1 cache misses and overall performance. As defined in Section 4, *fill-up_t* sets the threshold for the initial fill-up period for an L1-I cache, during which instructions are brought in until the cache is almost *full*. When the miss counter (*MC*) is lower than *fill-up_t*, a thread is not allowed to migrate. *Matched_t* sets the minimum number of tags that should be found on a remote cache before a thread migrates to it. Larger *matched_t* limits migration, while smaller values trigger too frequent migrations. *Dilution_t* is the minimum number of misses in the last 100 accesses to allow migration. It tends to restrict migration to the cases when more frequent misses are observed by a thread. The parameter choices could be thought of as a 3D space. To simplify, we first keep *dilution_t* value at zero, and explore the parameter space of *fill-up_t* and *matched_t*. In addition, we assume zero-overhead to search for remote tags. We later model an actual search mechanism.

Figure 7 reports I-MPKI, D-MPKI and performance relative to the baseline as a function of *fill-up_t* and *matched_t*. The *fill-up_t* values shown correspond to fractions of the L1-I cache capacity (512 cache blocks): ¼, ½, ¾, and *one*. The *matched_t* range shown is $2-10$; larger *matched_t* values further degrade performance. SLICC reduces instruction misses and increases data misses. Since instruction stalls, for OLTP workloads, account for 70% of overall cycle stalls [28], reducing instruction misses has a major effect on performance.

The results show that SLICC is not sensitive to different values of *fill-up_t*. *Fill-up_t* is actually a proxy for warming-up the caches; it affects only the first migration from a core. Thus with more migrations, the effect of *fill-up_t* diminishes. TPC-C and TPC-E transactions have large instruction counts and migrations. Figure 7 demonstrates that for *matched_t* values larger than four, performance benefits drop. On the other hand, although the overall MPKI at two is lower, performance at four is higher due to fewer migrations, and thus, lower overhead.

Next, we explore the parameter space of *dilution_t*. Using a small value for *dilution_t* triggers more frequent migrations. Using too large a value for *dilution_t* reduces migration overhead, but with a possible I-MPKI increase since it results in partial cache thrashing. Figure 8 shows L1 MPKI and relative to the baseline performance for *dilution_t* values 1 through 30 when *fill-up_t = 256* and *matched_t = 4* (best configuration from Figure 7). As *dilution_t* increases, instruction misses are reduced improving performance up to a point. Afterwards, larger *dilution_t* leads to fewer migrations, lesser overhead, but higher I-MPKI. There is a tradeoff between reducing instruction

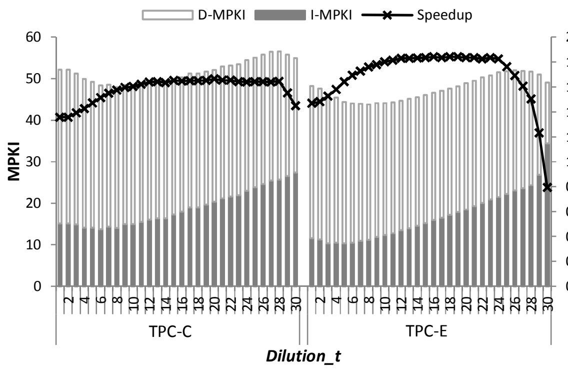

Figure 8: MPKI and relative performance as a function of *dilution_t*.

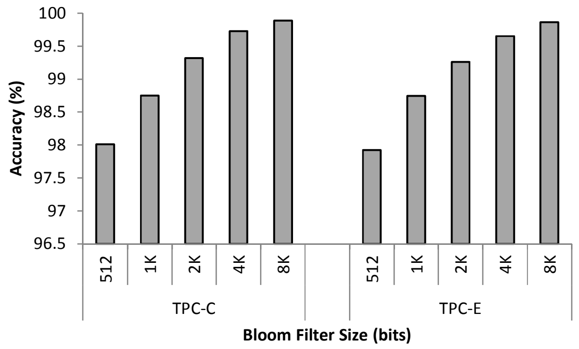

Figure 9: Partial-address bloom filter accuracy.

misses, and reducing migration overhead. Beyond *dilution_t* values of 28 (TPC-C) and 24 (TPC-E), although the overall MKPI is reduced, the performance degrades due to more limited migration. At even higher *dilution_t* values, migrations seize and performance drops below the baseline (for SLICC-SW, teams of transactions are injected to start on the same initial core, thus when migration stops some cores are underutilized).

In the remaining parts of this evaluation, we use *dilution_t = 10*, *fill-up_t = 256* and *matched_t = 4*. The last parameter means that the MTQ needs to keep track of only the four most recent misses, and that remote cache segment searching only requires finding where those four blocks are cached.

5.3. Cache Signature Accuracy

Section 4.2.3 explained that using a partial-address bloom filter reduces the overhead of remote cache segment searching. Figure 9 shows the accuracy of bloom filters of different sizes. The smallest bloom filter requires 512 bits to support evictions for a 32KB cache, with 64B blocks, and 512-sets. Accuracy is measured for all cache accesses and an access is accurate if the bloom filter and the cache agree on whether this is a hit or a miss. The trend is similar for TPC-C and TPC-E. In the rest of this paper, we experiment with 2K-bits filters as their effect on performance is less than 0.5% (99.3% accuracy).

5.4. Instruction Miss Change

Having determined a good SLICC configuration, this section shows that the three SLICC variants are able to reduce instruction misses much more than they increase data misses. Figure 10 shows the L1 I-MPKI for the baseline, SLICC, SLICC-SW, and SLICC-Pp. For MapReduce, since the instruction footprint fits in a 32KB cache, SLICC does not affect instruction or data misses.

Focusing on the other workloads, SLICC-SW reduces I-MPKI more than SLICC or SLICC-Pp. Compared to the baseline, SLICC-SW reduces I-MPKI by 56% and 61% for TPC-C and TPC-E, respectively. I-MPKI reductions are slightly lower with SLICC-Pp, more so for TPC-E than TPC-C, for the following reason: SLICC-Pp devotes one core to preprocessing. Given the transaction mix and transaction footprint sizes, having that extra core can be more important. All three variants of SLICC are sometimes forced to overcommit the caches by concurrently running transactions whose aggregate footprint does not fit on the total available L1 cache capacity. When overcommitting the caches, it is best to use stray threads since little opportunity for instruction reuse is lost when overcommitting with stray threads (a stray thread by definition is one that has few, if any, other ready threads sharing the same footprint). Overcommitting with non-stray threads happens more often for TPC-E than TPC-C partly because only 3% of TPC-E threads are stray compared to 12% of TPC-C threads. Furthermore, the need for stray threads is higher for TPC-E than TPC-C; SLICC spreads the transactions of TPC-E across $8-10$ cores, while TPC-C's transactions are spread across up to 14 cores.

SLICC improves I-MPKI less than the thread-type-aware alternatives, as it has to predict which is the next segment a thread will execute and where that segment currently is, using only a small preamble of the segment. The difference in reduction is more pronounced for TPC-C than TPC-E. TPC-C's overall instruction footprint is larger, resulting in higher variability in the instruction stream. Nevertheless, SLICC reduces instruction misses by 40.5%, on average.

As a comparison of the results for TPC-C-10 to those for TPC-C-1 shows, I-MPKI reductions persist mostly unaffected with the larger database.

5.5. Data Miss Change

During thread migration from core-A to core-B, three possible scenarios lead to extra data misses that would not have occurred otherwise: (1) a thread may read data on core-B that it fetched on core-A (extra misses on core-B for the same data blocks), (2) data writes on core-B to blocks fetched on core-A lead to invalidations that would not have occurred without migration (extra misses on core-B and invalidations on core-A), and (3) when a thread returns to core-A, it may find that data it originally fetched has since been evicted by another thread, or invalidated by itself (extra miss on core-A). Section 5.6 shows that instruction misses are more expensive than data misses performance-wise. Most data misses that result from migrations are served on-chip, allowing out-of-order execution to mostly absorb their latency.

Figure 10 reports the D-MPKI for all three SLICC variants and shows that SLICC-SW incurs an increase in D-MPKI of 11%, 1% and 4% over the baseline for TPC-C-1, TPC-C-10 and TPC-E, respectively. The other two variants exhibit a similar trend in D-MPKI increase. There is less locality and sharing in the larger data set of TPC-C-10, reducing the D-MPKI overhead when migrating.

Most of the increase in D-MPKI is for stores, which form 45% of total memory accesses, while loads are nearly unaffected. Due to slightly fewer migrations, SLICC-Pp increases D-MPKI less than SLICC-SW. As expected, SLICC is worse with an average D-MPKI increase of 9%.

We examined data prefetching to mitigate the increase in data misses. For each thread, we recorded the tags of the last *n*-referenced data blocks and then prefetched those blocks to the core the thread migrated to. This prefetcher did not improve performance, and past a

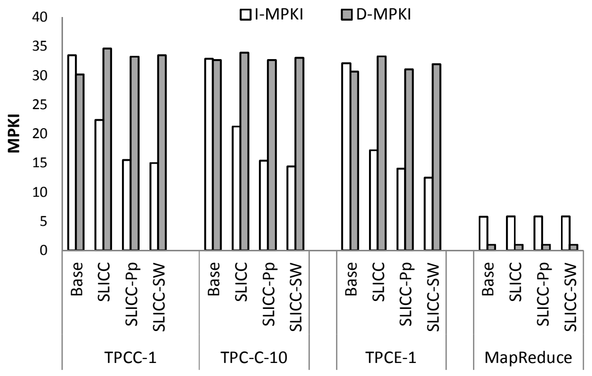

Figure 10: L1 I- and D-MPKI.

value of *n*, it hurts performance. There are several reasons why this prefetching proved ineffective. (1) The prefetched data increased the bandwidth on lower cache levels, which affects overall performance when *n* is high. (2) When *n* is low, there was not enough reuse. (3) Not all prefetched blocks are referenced again. (4) Finally, since 45% of the data accesses are stores, prefetching causes invalidations that would not have occurred otherwise.

This section showed that all SLICC variants improve I-MPKI significantly with a minor increase in D-MPKI, which is negligible with the larger database for TPC-C. These results suggest that SLICC can improve performance if, as expected, instruction cache misses degrade performance more than data cache misses; data misses can be partially overlapped with out-of-order execution. If this is the case, SLICC has the potential to offer a better balance of instruction *vs.* data cache misses over a conventional architecture.

Similar to D-MPKI, D-TLB misses increase on average by 11% and 8% with SLICC and SLICC-SW, respectively. I-TLB misses are within +/- 0.5% of the baseline.

5.6. Performance

This section reports the overall performance of SLICC relative to the baseline, a next-line instruction prefetcher, and a state-of-the-art prefetcher, PIF [5]. Figure 11 shows a 1.6× and 1.79× performance improvement over the baseline for SLICC-SW, on TPC-C-1 and TPC-E, respectively. On average, SLICC-SW and SLICC improve performance by 1.64× and 1.52× over the baseline, and 1.43× and 1.29× over a next-line instruction prefetcher.

Ferdman *et al.* report that PIF has nearly perfect coverage of L1-I misses [5]. Thus, we model an upper bound for PIF using a 512KB cache, with the delay of a 32KB cache. PIF's storage requirements are ∼40 KB per core. For TPC-C, SLICC-SW is within 2% of PIF's performance, with only 2.4% of PIF's storage requirements (see next section) per core. For TPC-E, SLICC-SW outperforms PIF by 21%. SLICC's speedup compared to PIF (and also the baseline) is a result of intelligent thread scheduling. Current schedulers assign threads to cores irrespective of their inherent-locality. Thus, even for larger caches, multiple similar threads run concurrently on different cores, each observing its own set of misses, for the same cache blocks. By pipelining similar threads, SLICC increases temporal inter-thread locality, hence it decreases overall miss rate observed by multiple threads.

MapReduce, which has an instruction footprint that fits in the L1-I cache, remains practically unaffected with SLICC.

5.7. Hardware Cost

Table 3 details the cost of all SLICC's hardware components. Section 4.2 described some of the components. In addition, SLICC

Table 3: Hardware Component Storage Costs.

Cache Monitor Unit	
Missed-Tag Queue (MTQ)	60-bits (16-core, *matched_t* = 4)
Miss Shift-Vector (MSV)	100-bits
Cache Signature (Bloom Filter)	2K-bits
Total	2208 bits (276 Bytes)
Thread Scheduler	
Thread Queue	30-entries (12-bits numerical ID, 48-bits pointer to thread context 4-bits core ID)
Total	1920 bits (240 Bytes)
Team Formation (SLICC-SW & SLICC-Pp)	
Team Management table	60-entries (12-bits numerical ID, 32-bits timestamp, 4-bits type ID, 4-bits team ID, 8-bits team index)
Total	3600 bits (450 Bytes)
Grand Total	7728 bits (966 Bytes)

requires a *thread queue* that holds threads waiting for cores. Each entry contains a unique numerical ID, a pointer to the threads' context, and a core ID. The thread queues can be local to each core, or centralized to one core. The table shows the cost for a centralized queue. Fewer entries are required when the queues are local to each core. The *team management table* is responsible for forming teams of similar threads (not required by SLICC). Each entry consists of: a unique numerical ID, a type ID, a team ID, index within a team, and a timestamp. The team management table is best thought of as being centralized, since every core needs to know which cores are assigned to which teams. We can either have one centralized copy, or per core copies that are kept coherent. For this work we simulated a centralized copy at one of the cores and modeled the necessary traffic.

On each core, a SLICC agent is responsible for managing the thread queue. The thread queue is a circular FIFO buffer and the first entry is executed until it migrates, completes, or gets blocked for I/O. On the latter case, the thread is moved to the end of the queue. With an over-provisioned thread queue of 30 threads, and a copy of the team management table, per core, SLICC requires a maximum of 966 bytes in addition to logic. All logic operations for SLICC are not on the critical path.

5.8. Remote Cache Segment Search Activity

As per the description of Section 4.2.3, a thread that wants to migrate has to find which cache, if any, holds the next code segment. Our results, thus, far modeled this searching by including separate messages for the corresponding miss messages using *separate* broadcasts. We do so to obtain an upper limit of the overhead these messages may induce. We report the frequency of these messages as Broadcasts per Kilo Instructions (BPKI) and find that it is very low. For TPC-C, BPKI is 2.204 for SLICC and 0.28 for SLICC-SW and SLICC-Pp. For TPC-E, BPKI is 1.328 for SLICC and 0.367 for SLICC-SW and SLICC-Pp. As Section 4.2.3 explained these requests are required

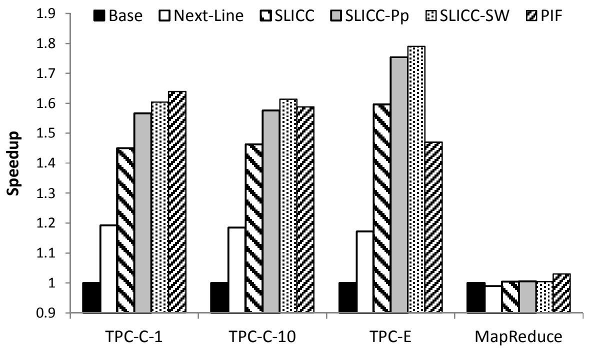

Figure 11: Performance.

anyhow for normal miss processing. The ownership information required by SLICC is either already available or should be possible to piggyback on existing responses.

6. Related Work

Instruction prefetching solutions have evolved from simple stream buffers [14, 26] to highly accurate, sophisticated stream predictors [6, 5]. Accurate prefetchers for OLTP are expensive, requiring $\sim$40KB of extra storage per L1 cache. Since these prefetchers track execution sequences, their storage requirements should increase with the instruction footprint. In addition, they neglect the possible presence of idle cores, and do not avoid code and prediction redundancy, under-utilizing on-chip resources. In this work we compared SLICC to PIF [5], a state-of-the art prefetcher that achieves near optimal instruction miss coverage. We showed that SLICC was able to achieve 98% of PIF's performance for TPC-C, using only 2.4% of the storage area overhead, while outperforming it by 21% for TPC-E.

Chakraborty *et al.* show a high-degree of redundancy in instruction fragments across threads concurrently running on multiple cores [3]. They propose CSP, which employs thread migration to distribute the dissimilar instruction code segments and group the similar ones together. For system code, which is commonly used by multiple threads, CSP fragments and distributes the code across a group of dedicated cores. CSP then migrates threads to these dedicated cores to execute system code. When threads are done, they return back to their original cores to resume execution for the user-level code. Thus CSP is limited to fragmenting OS code, losing opportunities of fragmentation within user code. SLICC generalizes thread migration to include interleaved user-OS code fragmentation points. In addition, thread migration in SLICC is managed by the hardware, while with CSP, the OS performs the migrations.

Atta *et al.* suggested using thread migration for reducing instruction misses in OLTP and demonstrated its potential to reduce I-MPKI without presenting a solution [2]. No performance analysis was conducted. This work presents a working solution and demonstrates its performance benefits.

STEPS [9] aims to minimize instruction misses from the software side. Like SLICC, it groups threads executing similar transactions into teams. It, either manually or by using a profiling tool, breaks each transaction's instruction footprint into smaller instruction chunks in a way that each chunk can fit in the L1-I cache. Then, all the threads in the same team execute the first chunk, rather than executing the whole transaction without any interruption, on the same core by context switching to the other thread when one completes the execution of the chunk. STEPS repeats this process for all the chunks, allowing instruction re-use across many threads for each chunk. SLICC exploits the same way of re-using the instructions already brought into the cache by previous threads. However, rather than context-switching on the same core, SLICC migrates threads to another core so that they can continue their execution. Moreover, SLICC dynamically detects the synchronization points in a transaction rather than using a priori manual or profiling based software instrumentation. Future work may look at combining the time-domain pipelining of STEPS with the space-domain pipelining of SLICC.

Data-oriented transaction execution (DORA) indirectly affects the instruction footprint of a transaction [21]. It divides a transaction into smaller actions based on the data being accessed at a particular transaction part. Then, each of those actions are sent to their corresponding worker threads, reducing the overall number instructions executed by a single thread per transaction. Such a design might not necessarily break a transaction into instruction parts that can fit in L1-I. However, if combined with SLICC, it can give better hints on where to migrate or reduce the total number of migrations needed to be done per worker thread.

Hardavellas *et al.* [8] observe that more than 60% of a distributed shared L2 accesses are for instructions. They adapt a NUCA block placement policy according to workload categorization, and allow replication of (read-only) instructions, which shortens the distance between L1-I caches and L2s. This reduces the L1-I miss penalty, but does not reduce the miss rate.

Other recent thread migration proposals target power management, data cache, or memory coherence [19, 25, 16, 27].

7. Conclusions

Literature showed that memory stalls for OLTP workloads account for 80% of their execution time, and L1 instruction misses account for 70-85% of overall stall cycles. We corroborate these results and show that 94% of L1 capacity misses are for instructions. Additionally, we show that recently proposed replacement policies, which reduce miss rates for some workloads, leave a lot of room for improvement compared to using larger L1-I caches. Previous works tackle this problem in software or hardware, but they are either impractical (require code instrumentation) or relatively expensive (large on-chip data structures).

This work presented a solution based on thread migration, SLICC. Similar to CSP [3] and STEPS [9], we exploit the code commonality observed across multiple concurrent threads. Unlike CSP, we do not limit code reuse to OS code segments. Unlike STEPS, instead of context switching on the same core, we distribute the instruction footprint across multiple cores and migrate execution. SLICC is a low-level hardware algorithm that requires no code instrumentation and efficiently utilizes available cache capacity, by improving intra- and inter-thread locality.

SLICC reduces the instruction misses for OLTP by 56% on average at the expense of an 5% average increase in data misses. SLICC improves the overall performance by 68% on average over the baseline and performs better when the input database is larger. Compared to a state-of-the-art instruction prefetcher (PIF), SLICC improves performance by 21% for TPC-E and comes within 2% for TPC-C, with only 2.4% of relative area overhead. When tested on MapReduce, a cloud workload that has a relatively small instruction footprint, SLICC was robust and did not affect the L1 miss rates or performance.

8. Acknowledgments

We thank the members of the AENAO and DIAS laboratories, Adrian Popescu, the reviewers, and Jared Smolens for their comments and help. We thank Sudhakar Yalamanchili, Jun Wang, and the whole Georgia Tech development team for providing us with the Zesto simulator. This work was partially supported by an NSERC Discovery grant, an NSERC CRD with IBM, a Sloan research fellowship, NSF grants CCR-0205544, IIS-0133686, and IIS-0713409, an ESF EurYI award, and Swiss National Foundation funds.

References

[1] Advanced Micro Devices, "Secure virtual machine architecture reference manual," May 2005.

[2] I. Atta, P. Tözün, A. Ailamaki, and A. Moshovos, "Reducing OLTP instruction misses with thread migration," in *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, 2012, pp. 9–15.

[3] K. Chakraborty, P. M. Wells, and G. S. Sohi, "Computation spreading: employing hardware migration to specialize CMP cores on-the-fly," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 283–292.

[4] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012, pp. 37–48.

[5] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 152–162.

[6] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Temporal instruction fetch streaming," in *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, 2008, pp. 1–10.

[7] Gartner, "Market share: Database management system software, worldwide, 2008," 2009, available at http://www.gartner.com/DisplayDocument?id=1044912.

[8] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: near-optimal block placement and replication in distributed caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009, pp. 184–195.

[9] S. Harizopoulos and A. Ailamaki, "Improving instruction cache performance in OLTP," *ACM Transactions on Database Systems*, vol. 31, no. 3, pp. 887–920, Sep. 2006.

[10] M. D. Hill and A. J. Smith, "Evaluating associativity in CPU caches," *IEEE Transactions on Computers*, vol. 38, no. 12, pp. 1612–1630, Dec. 1989.

[11] IBM, "IBM breaks double digit performance barrier with 10 million transactions per minute," 2010, available at http://www-03.ibm.com/press/us/en/pressrelease/32328.wss.

[12] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010, pp. 60–71.

[13] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi, "Shore-MT: a scalable storage manager for the multicore era," in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, 2009, pp. 24–35.

[14] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 364–373.

[15] K. Keeton, D. Patterson, Y. Q. He, R. Raphael, and W. Baker, "Performance characterization of a Quad Pentium Pro SMP using OLTP workloads," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998, pp. 15–26.

[16] M. Lis, K. S. Shim, M. H. Cho, O. Khan, and S. Devadas, "Directoryless shared memory coherence using execution migration," in *Proceedings of the 24th IASTED International Conference on Parallel and Distributed Computing and Systems*, 2011.

[17] G. H. Loh, S. Subramaniam, and Y. Xie, "Zesto: A cycle-level simulator for highly detailed microarchitecture exploration," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 53–64.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming language design and implementation*, 2005, pp. 190–200.

[19] P. Michaud, "Exploiting the cache capacity of a single-chip multi-core processor with execution migration," in *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, 2004, pp. 186–.

[20] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," HP, Tech. Rep., 2009.

[21] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki, "Data-oriented transaction execution," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 928–939, Sep. 2010.

[22] PARSA, "Data analytics benchmark with hadoop mapreduce framework," 2012, available at http://parsa.epfl.ch/cloudsuite/analytics.html.

[23] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai, "Bloom filtering cache misses for accurate data speculation and prefetching," in *Proceedings of the 16th International Conference on Supercomputing*, 2002, pp. 189–198.

[24] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007, pp. 381–391.

[25] K. K. Rangan, G.-Y. Wei, and D. Brooks, "Thread motion: fine-grained power management for multi-core systems," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009, pp. 302–313.

[26] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso, "Performance of database workloads on shared-memory systems with out-of-order processors," in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1998, pp. 307–318.

[27] K. S. Shim, M. Lis, O. Khan, and S. Devadas, "Judicious thread migration when accessing distributed shared caches," in *Proccedings of the Third Computer Architecture and Operating System Co-design*, 2012.

[28] P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki, "From A to E: Analyzing TPC's OLTP Benchmarks – The obsolete, the ubiquitous, the unexplored?" EPFL, Tech. Rep., 2012.

[29] TPC, "TPC benchmark C (OLTP) standard specification, revision 5.11," 2010, available at http://www.tpc.org/tpcc.

[30] TPC, "TPC benchmark E standard specification, revision 1.12.0," 2010, available at http://www.tpc.org/tpce.

[31] TPC, "TPC-C ten most recently published results," 2012, available at http://www.tpc.org/tpcc/results/tpcc_last_ten_results.asp.

[32] TPC, "TPC-E ten most recently published results," 2012, available at http://www.tpc.org/tpce/results/tpce_last_ten_results.asp.

[33] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel Virtualization Technology," *IEEE Computer*, pp. 48–56, 2005.