Reducing OLTP Instruction Misses With Thread Migration

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

[‡]École Polytechnique Fédérale de Lausanne [†]University of Toronto

ABSTRACT

During an instruction miss a processor is unable to fetch instructions. The more frequent instruction misses are the less able a modern processor is to find useful work to do and thus performance suffers. Online transaction processing (OLTP) suffers from high instruction miss rates since the instruction footprint of OLTP transactions does not fit in today's L1-I caches. However, modern many-core chips have ample aggregate L1 cache capacity across multiple cores. Looking at the code paths concurrently executing transactions follow, we observe a high degree of repetition both within and across transactions. This work presents *TMi* a technique that uses thread migration to reduce instruction misses by spreading the footprint of a transaction over multiple L1 caches. *TMi* is a software-transparent, hardware technique; *TMi* requires no code instrumentation, and efficiently utilizes available cache capacity. This work evaluates *TMi*'s potential and shows that it may reduce instruction misses by 51% on average. This work discusses the underlying tradeoffs and challenges, such as an increase in data misses, and points to potential solutions.

1. INTRODUCTION

Online transaction processing (OLTP) workloads are memory bound spending 80% of their time stalling for memory accesses [10]. Most of these stalls are due to first-level instruction cache misses [3]. Previous works tackle this problem in software [6] or hardware [9, 15, 2, 4].

Traditional OLTP systems randomly assign transactions to worker threads, each of which runs on one core of a modern multi-core system. As we also confirm, the instruction footprint of a typical OLTP transaction does not fit into a single L1-I cache, resulting in a high instruction miss rate. Enlarging the L1-I cache is not a viable solution; to avoid impacting the CPU clock frequency, the L1 caches of virtually all high-performance processors today are limited to about 32KB, despite the growing size of L2 and L3 caches. However, this work demonstrates that the footprint of a typical OLTP transaction would comfortably fit in the aggregate L1-I cache capacity of modern many core-chips. There is then an opportunity to reduce instruction misses by spreading the footprint of transactions over multiple L1-I caches provided that there is sufficient code reuse. Fortunately, in OLTP, there is a high-degree of instruction footprint reuse both within a transaction and across multiple, concurrently running transactions [2, 6].

This paper investigates *TMi*, a dynamic hardware solution that uses thread migration to minimize instruction misses for OLTP applications. *TMi*, spreads each transaction over multiple cores, so that each L1-I cache holds part the instruction footprint. *TMi* relies upon: (1) a hardware scheduler knowing the thread types, and (2) having a fast and efficient mechanism to determine whether a set of cache blocks (tags) exist in a given cache. *TMi* exploits intra- and inter-thread footprint reuse as follows: (1) If a thread loops over multiple code parts that are spread over multiple caches, the observed miss rate will be lower since in a conventional cache each part would evict the others resulting in thrashing, (2) The first thread of a particular type effectively prefetches and distributes the common instruction footprint for the rest. TMi differs from past OLTP instruction miss reduction techniques in that it a pure hardware, low-level solution that avoids undesirable instrumentation, utilizes available core and cache capacity resources, and requires no changes to the existing code or software system. TMi is not free of tradeoffs and challenges. Migrating threads takes times and results in more data misses and hence TMi must balance the cost of these overheads against the benefit gained from reducing instruction misses.

This work makes the following contributions:

- It analyzes instruction and data footprints for OLTP workloads (TPC-C and TPC-E [17]) and reaffirms that they suffer from instruction misses; on average 91% of L1 cache capacity misses are instruction misses.
- It demonstrates that the recently proposed replacement policies [14], which can reduce miss rates significantly for some workloads, are not effective for OLTP workloads (less than 6% reduction).
- It identifies the requirements and challenges for an ideal thread-migration-based solution.
- It presents and evaluates TMi, a practical thread migration algorithm. It shows that TMi can reduce instruction misses by 51% on average.
- It identifies the challenges associated with thread migration, namely data misses and migration overhead.

The remaining of this document is organized as follows.

Proceedings of the Eight International Workshop on Data Management on New Hardware (DaMoN 2012), May 21, 2012, Scottsdale, AZ, USA
Copyright 2012 ACM 978-1-4503-1445-9 ...$10.00.

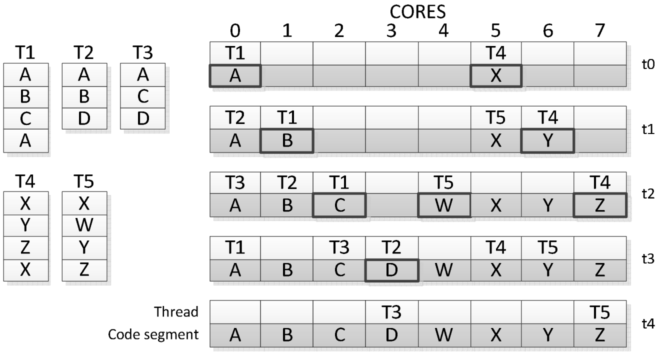

Figure 1: Example to illustrate thread migration and reuse of common code segments. Left: T1-3 and T4-5 are threads running similar transactions. Right: 8-core system. The shaded area is the cache activity (thick border = warm-up phase).

Section 2 analyzes the nature of the problem and the requirements for an ideal solution. Section 3 describes the proposed automated thread migration algorithm, TMi. In Section 4 we present the evaluation results. Section 5 surveys the related work. Section 6 discusses challenges and practical considerations. Finally, Section 7 concludes.

2. PROBLEM & OPPORTUNITY

In typical multi-threaded OLTP systems, a single thread handles a single transaction and multiple threads executing the same transaction type usually run concurrently. Individual threads, with memory footprints that do not fit in the L1-I cache, suffer from high miss rates. For the examined OLTP workloads, Ferdman *et. al.* [3] show that memory stalls account for 80% of the execution time. Section 4 shows that instruction capacity misses account for 83% of total L1 cache misses.

Although similar transactions (threads) vary in their control flow, they have common code segments, leading to 80% redundancy in L1-I caches across multiple cores [2]. We exploit the availability of many cores and multiple concurrent threads with inherent code commonality. Using thread migration, we aim to increase the reuse of instruction blocks brought in the cache. We propose a hardware scheduling algorithm that divides and distributes the instruction code footprint across multiple cores. The algorithm dynamically pipelines and migrates threads to cores that are predicted to hold the code blocks to be accessed next.

Figure 1 demonstrates thread migration using an example. On the left, there are five threads T1-T5 running on 8-cores, where T1-T3 and T4-T5 execute similar transactions, with slightly different instruction streams. The instruction footprint of T1 is 3× larger than L1-I cache size. It executes the following code segments in order: A-B-C-A, where each segment fits in the L1-I cache. T2 and T3 are of the same type as T1, hence they share common code segments with T1, but they are not identical. A typical system would assign T1, T2, and T3 to separate cores, and since their footprints do not fit in the L1-I cache, each individual thread would suffer instruction misses.

The ideal scenario for thread migration would be as follows: initially (at time t0), T1 starts execution on core-0. When all cache blocks for A are brought in the cache, T1 migrates to core-1 (at t1), where it continues executing filling the cache with blocks from B. T2 is then scheduled to start execution on core-0 (at t1), ideally observing hits for all blocks in A. At a higher level, the process continues and T1 warms-up caches 0, 1, and 2 with A, B, and C, respectively. If T2 and T3 where to follow the same path as T1 then they would experience no misses.

The threads do not need to follow identical paths for migration to be beneficial. Code segment D illustrates this. Since code segment D, accessed by T2, was not part of T1's instruction stream, T2 needs to warm-up core-3 and suffer some extra misses. Provided that T3 executes after T2 finishes filling the cache with D, T3 does not suffer any instruction misses. At t3, when T1 goes back to A, it gets rescheduled to core-0. T4-T5 exhibit similar behavior, but they get assigned to a different set of cores, avoiding conflict with T1-T3. The same process applies for all threads that arrive later: if part (or all) of their code segments exist in the L1-I cache, they migrate to the appropriate core and do not miss on these segments.

We observe that transactions vary in their control flow, hence we should not impose any specific pipelining; i.e., we do not restrict similar threads to follow the exact same path. Thread migration should dynamically detect: (a) *When* a thread should migrate (i.e., when is the cache *full*)? (b) *Where* to migrate (i.e., which cache holds the code segment we start to touch, if any)? Thus we need to maintain information about threads and caches to be able to make judicious migration decisions.

3. AUTOMATED THREAD MIGRATION

With the requirements we set in Section 2 in mind, here we detail our thread migration algorithm, TMi, and its challenges. Our main focus in this paper is the potential of thread migration in reducing instruction misses for OLTP. Hence, the naïve TMi as described here might seem inefficient. However, we discuss why we believe a practical implementation is possible in Section 3.2 and Section 6.

3.1 Migration Algorithm

The presented TMi design assumes it knows the *type* of each transaction so that it can group same type threads into teams. We discuss possible methods to detect transaction types in Section 6. TMi schedules each team without preemption. Threads within a team are independently migrated among cores without any central control. Migration decisions are made based on a set of rules and inputs.

Initially, the system operates normally and no migration takes place. First, TMi must detect when the L1-I cache of a given core becomes *FULL*, that is filled with enough instructions so it is better for the thread to migrate to another core to avoid thrashing. To detect a *FULL* cache, TMi uses a resettable miss counter (*MC*) and a full miss-threshold (*FMT*). When *MC* exceeds *FMT*, TMi enables migration.

Migration does not happen immediately after a cache becomes *FULL*. The thread may immediately loop back to the same code segment or may temporarily follow a somewhat different path.

When migration is enabled, we need a mechanism to select a good target cache to migrate to. Ideally, we would migrate to a cache that has the instructions that this thread will execute next. TMi records recently missed tags in a FIFO

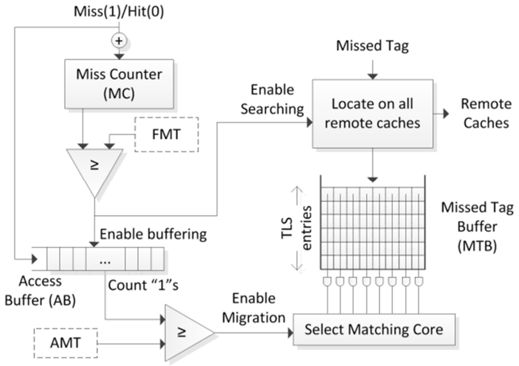

Figure 2: TMi organization.

missed-tags buffer (*MTB*). We refer to the size of *MTB* as the tag list size (*TLS*). For a given thread on a given core, if the *MTB* is full, TMi searches all remote L1-I caches for the missed tags, and makes one of the following decisions:

- If there is a match in a remote cache for *TLS* tags, TMi migrates the thread to the matched core.
- Else, the thread is migrated to an idle core (if any).
- Else, do not migrate.

An additional condition is required to restrict migration to the cases when a thread starts to miss more frequently. In a sub-optimal scenario, threads have to miss for a few tags before migrating (*TLS* tags must be located on a remote cache). This can lead to eviction of useful cache blocks that were fetched in the warm-up phase, creating gaps in the instruction stream. TMi mitigates this by not allowing threads to migrate for occasional misses that fill-up these gaps. Access buffer (*AB*) is a 100-bit FIFO queue recording the history of the last 100 cache accesses (enabled when cache is *FULL*). A logic-0 and logic-1 represent a cache hit and miss, respectively. When the number of logic-1 bits reach a threshold (*AMT*), TMi enables migration. If *AMT = 0*, then TMi neglects this condition.

TMi resets *MTB* and *AB* with every migration. When a team of threads completes execution, TMi resets all *MC*s, *MTB*s and *AB*s.

Figure 2 describes the thread migration decision process. TMi components are: (a) Miss counter *MC*. (b) Access buffer *AB* is a 100-bit FIFO. (c) Missed Tag Buffer *MTB* is a FIFO of n-bit entries, where n is the number of cores. (d) Full miss threshold *FMT* determines when a cache is *FULL*. (e) Access miss-threshold *AMT* is the minimum number of misses in a window of 100 accesses to enable migration. *MC*, *AB* and *MTB* are resettable.

3.2 Migration Cost & Challenges

Tag Search. We believe it is possible to avoid individual tag searches (e.g., keeping track of signatures as opposed to a complete tag list) and searching all cores (e.g., serial node searching, history-based candidate node prediction).

Data Misses. Migrating across cores may increase the data miss rate; data that might be reused is left behind during a migration. Experiments show that the data miss rates increase significantly.

Existing architectures simply use caches to reduce instruction and data misses. No additional effort is made to balance the relative cost of instruction vs. data misses. A relevant question is whether the relative costs are appropriately balanced in modern systems. TMi facilitates balancing the relative costs of the two types of misses. There are reasons to believe that instruction misses, up to a point, impact performance more than data misses. For example, modern architectures use instruction level parallelism to hide the impact of data misses. Without instructions these ILP techniques remain ineffective. However, this has to be proven through experimental evaluation, and is left for future work.

Simultaneous Multithreading (SMT) is another way for reducing instruction stalls but it requires extra resources. Additionally, if threads are not synchronized, it increases pressure on instruction caches leading to more thrashing.

We analyze data misses and discuss potential solutions to mitigate their impact in Section 4.5.

Migration Overhead. Fast context switching is attainable in software [6] and we argue that the same is possible for the hardware. Estimating hardware migration overhead is out of the scope of this paper, but in Section 4 we seek to increase the number of instructions between migrations, thus reducing the relative importance of the context switching cost.

Overhead of data-structures used. All data structures described here are per core. *MC* exists in the Performance Monitoring Unit (PMU) of most modern processors. *MTB* is a list of *TLS* entries and we consider two implementations: In the first, an entry is simply a tag. A more compact and faster list consists of n-bits per entry, where n is the number of cores. A logic-1 bit at index i, of tag j, means that the cache of core-i holds a valid copy of tag j. By ANDing all bits mapped to core-i, we know whether it holds all tags, or not. This mechanism relies on the coherence protocol to identify shared cache blocks, and makes faster migration decisions. Figure 2 shows the latter implementation. *AB* is a 100-bit FIFO queue used to record the history of cache accesses. A *thread queue* holds a 12-bit identifier and context state. We assume virtualizing this information on lower cache levels [1]. Note that all logic operations for TMi are not on the critical path.

4. EVALUATION

Current real and full system simulation platforms 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 use trace simulation. We use PIN [12], an instrumentation tool that is able to inject itself in x86 binaries, to extract instruction and data access traces. Although PIN instruments only application level code, our proposal is generic and applies to system level code, too. Previous work shows that thread migration is beneficial for system level code [2].

The baseline configuration *NO-MGRT* assumes no thread migration and assigns threads to cores randomly. *TMi-ST* is the proposed thread migration algorithm, which groups *similar threads* into teams. We examine various thread migration parameters. Unless otherwise indicated, we simulate a 16-core system with 32KB 8-way set-associative instruction and data caches with 64B blocks, and LRU replacements.

We run TPC-C and TPC-E on top of a scalable open-source storage manager; Shore-MT [8]. Traces are extracted,

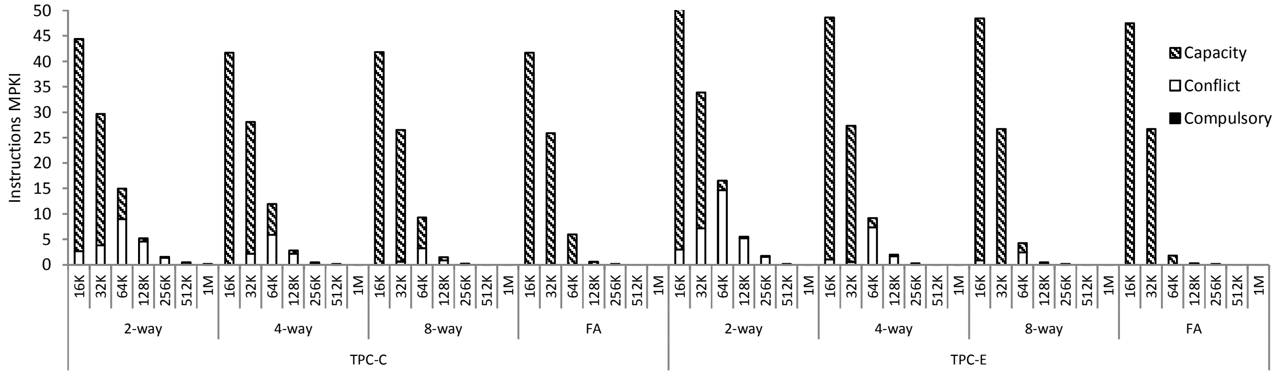

Figure 3: Breakdown of MPKI for different L1-I sizes and associativity (lower is better).

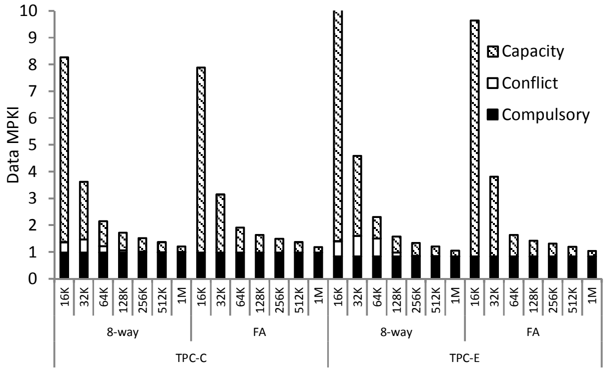

Figure 4: Breakdown of MPKI for different L1-D sizes and associativity (lower is better).

per transaction type, to form a mix of 1K threads, per benchmark. Teams of 20-30 similar threads are scheduled without preemption (threads are more than cores to consider thrashing). We use misses per kilo instructions (MPKI) as our metric for instruction (I-MPKI) and data (D-MPKI) misses. We use kilo instructions per migration (KIPM) as a migration overhead metric. Ignoring secondary effects, the higher the KIPM the less fraction of time is spent on migration. An ideal solution should minimize MPKI and maximize KIPM.

In our evaluation we address the following questions: (1) What are the L1 miss characteristics for OLTP workloads (Sections 4.1 and 4.2)? (2) What are good configuration parameters for TMi (Sections 4.3 and 4.4)? (3) How does TMi affect data misses (Section 4.5)? (4) What is the impact of a migration algorithm, which is unaware of thread types, on L1 misses (Section 4.6)?

4.1 Instruction and Data Footprints

This section examines the instruction and data footprints through an analysis of misses per kilo-instructions (MPKI) on full-associative (FA) and set-associative (SA) L1 caches. Hill and Smith [7] categorize cache misses into three Cs: *Capacity* misses result from limited cache capacity (relatively smaller caches may increase capacity misses), *Conflict* misses are those resulting from reduced associativity (smaller sets may increase conflict misses), and *Compulsory* misses are for the first reference to each unique data block (you can avoid if you prefetch). By identifying where most misses come from, we highlight the reasons behind the memory stalls; is it cache size, associativity, or bad locality?

Figures 4 and 3 has the breakdown of MPKI into the three Cs. Figure 3 shows that for FA caches (no conflict misses), the instructions fit in 128KB – 256KB caches. A 512KB cache incurs only compulsory misses (i.e., perfect cache). Today, most modern microprocessors use 32KB L1s. Thus, the full footprint can fit in 4 – 8 caches (or more).

For 32KB caches, instruction capacity misses are $9-12\times$ more than data capacity misses, while data compulsory misses are two orders of magnitude larger than instruction compulsory misses. We conclude that OLTP transactions have large instruction footprints with much lower locality than their data footprint. The instruction streams exhibit a recurring pattern (lots of reuse), which has a relatively long period, leading to eviction of useful blocks that are re-accessed.

These observations support TMi; it could avoid lots of instruction misses by virtually increasing the L1-I cache size per thread. In the rest of our analysis we use 32KB 8-way SA L1 caches. We do so as they typical in modern processors, e.g., the Intel ®Core $^{\text{TM}}$2 Quad.

4.2 Replacement Policies

Qureshi *et al.* identify that not all workloads are LRU-friendly [14]. They propose static (LIP, BIP) and dynamic (DIP) insertion policies that are effective in reducing miss rates significantly for some workloads. We measure the impact of these policies on OLTP workloads and compare them against LRU. In Figure 5, we witness less than 6% improvement for DIP on the instruction MPKI for the baseline setup. Nevertheless, these policies are orthogonal to TMi.

4.3 Tuning Migration Parameters

This section explores the effect of *FMT* and *TLS* on instruction misses (I-MPKI) and instruction count between migrations (KIPM). As defined in Section 3.1, *FMT* sets the initial warm-up for an L1-I cache. When the miss counter (*MC*) is lower than *FMT*, a thread is not allowed to migrate. *TLS* sets the minimal number of tags a remote cache should hold before a thread migrates to it. Larger *TLS* limits migration, while smaller *TLS* triggers too frequent migration; we are optimizing values for *FMT* and *TLS* to reduce I-MPKI and increase KIPM.

Figure 6 shows the results for TMi-ST. We examine *FMT* values of 128 – 1024 and *TLS* values of 2, 4, and 6. We report relative I-MPKI with respect to the baseline. Larger

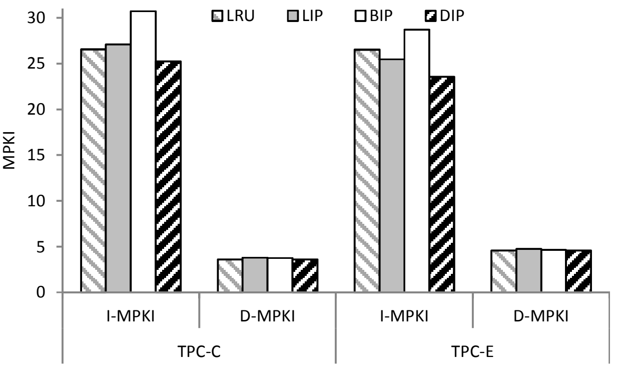

Figure 5: Effect of different replacement policies for OLTP workloads.

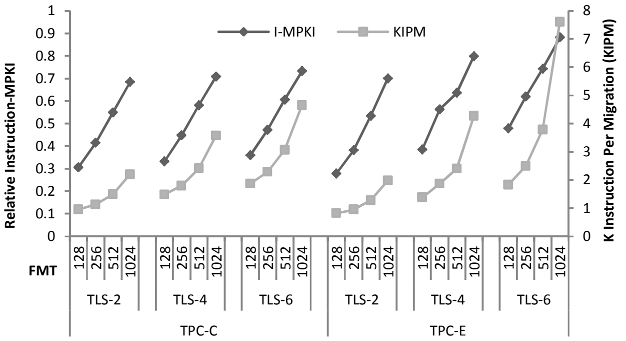

Figure 6: Exploring the parameter space of *FMT* and *TLS* values

TLS translates into fewer migrations, resulting in higher I-MPKI for all values of *FMT*, while smaller *TLS* reduces KIPM. TPC-E is more sensitive to larger *TLS* values than TPC-C (TPC-E has more non-uniform instruction streams). For *TLS* values larger than 6, TPC-E shows very few migrations with negligible I-MPKI improvement. *FMT* favors smaller values, assigning each cache a smaller working set and reducing conflicts. For 64B block size (512 blocks per cache) and *FMT* values larger than 512, cache blocks are definitely evicted in the warm-up phase.

To summarize, lower *FMT* and *TLS* reduce I-MPKI by 70%/73% for TPC-C/TPC-E, with slightly less than 1 KIPM. We conclude that a good combination would be 256/6 for *FMT*/*TLS*, resulting in 53%/38% I-MPKI reduction, and 2.3/2.5 KIPM, for TPC-C/TPC-E. The remaining of the evaluation section uses these values.

4.4 Access Miss Threshold

AMT restricts migration to the cases when a thread starts to miss more frequently. In a sub-optimal scenario, threads have to miss for a few tags before migrating (*TLS* tags must be located on a remote cache). This can lead to eviction of useful cache blocks that were fetched in the warm-up phase, creating gaps in the instruction stream. *AMT* mitigates this by not allowing threads to migrate for occasional misses that fill-up these gaps.

Figure 7 shows I-MPKI and KIPM for TMi-ST, with a range of *AMT* values (1 – 12). Using small *AMT* triggers more frequent migrations. Using too large *AMT* increases KIPM by reducing migrations, but with a possible I-MPKI increase since it results in more evictions. For TPC-C, *AMT* values 6 to 9 are better. TPC-E shows a non-uniform behavior for *AMT* values 9 (high I-MPKI) and 11 (low I-MPKI). This behavior could be a result of an irregularity in the instruction stream. We have noticed a direct correlation between *AMT* and *TLS* values: the point $TLS = AMT$ is a sweet-spot for I-MPKI and for larger *AMT*, KIPM increases linearly. Due to limited space, we do not show graphs for different *TLS* values. For the remainder of the evaluation section, we set *AMT* to eight (59%/43% I-MPKI reduction and 2.4/2.6 KIPM for TPC-C/TPC-E).

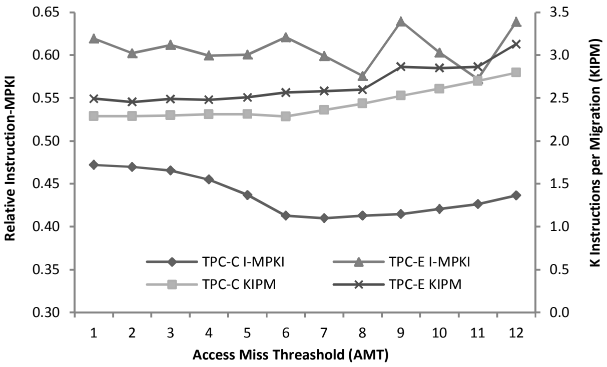

Figure 7: Exploring the parameter space of *AMT*

4.5 Data Misses

While reducing instruction misses, TMi increases data misses in three ways; (1) a thread may access data that it fetched on one core, when it migrates to another core, (2) when a thread returns to a core, it may find that data that it originally fetched has since been evicted by another thread, (3) data writes of T1 on core-B to blocks fetched on core-A lead to invalidations that would not have occurred without migration. Although we believe that instruction misses are more expensive than data misses (performance-wise), we have to account for D-MPKI.

Figure 8 shows that TMi-ST incurs an average increase in D-MPKI of 4× over the baseline. We notice that the increase in D-MPKI is the result of writes (WR D-MPKI), while reads are nearly unaffected. The average total MPKI is reduced by 4% over the baseline. For an inclusive cache hierarchy, all data misses can be served on-chip via lower levels of cache (L2/L3), allowing out-of-order execution to partially hide it. In addition, we believe that data misses could be mitigated by techniques like prefetching, especially when we have a strong clue on what to prefetch.

4.6 Blind Migration

TMi-ST assumes knowledge of thread types. We experiment with an algorithm (TMi-B) that migrates threads without considering their types. TMi-B utilizes an aggressive scheduler (1K concurrent non-similar threads), while TMi-ST forms teams of similar threads, where each team is scheduled without preemption. Figure 8 shows I-MPKI/D-MPKI for TMi-B. We notice a 31% average reduction in I-MPKI over the baseline, and an average increase of 40% over TMi-ST. Read D-MPKI increases by 2× on average, while write D-MPKI is slightly reduced. Closer inspection shows that blind migration creates large access periods be-

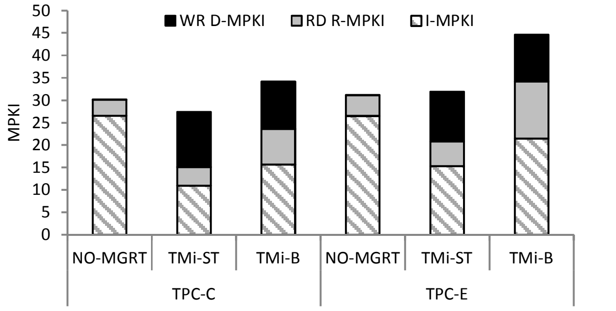

Figure 8: I-MPKI decrease vs. D-MPKI increase

fore a thread migrates back to a previous core, finding that most of its cache blocks are evicted. Conflicts occur when non-similar threads are assigned to the same core. We define *distance between similar threads* (DBST), as a metric to measure the conflict degree (lower is better). DBST is the distance between two similar threads executing on the same core. TMi-B and TMi-ST show an average DPST of 28.3 and 2.53, respectively. We conclude that identifying thread types is necessary for TMi to effectively reduce in I-MPKI.

5. RELATED WORK

STEPS [6] is a software solution that increases instruction reuse by grouping threads into teams of similar transactions. It breaks transaction instruction footprints into smaller cache chunks. Then instead of running a full transaction, it runs all transactions for the first chunk, then for the second and so on. STEPS identifies points in the code where context-switches should occur either manually or by using a profiling tool, which makes it hard to use in practice. TMi is similar to STEPS in the sense that we promote locality, but instead of context-switching on a single core, we migrate threads to other cores, utilizing available on-chip resources and reducing thread queuing delay. In addition, TMi does not require non-trivial manual intervention or instrumentation to detect synchronization points.

On the hardware side, instruction prefetching solutions have evolved from simple low accuracy stream buffers [9, 15] to highly accurate sophisticated stream predictors [5, 4]. Accurate prefetchers are expensive requiring ∼40KB of extra storage per L1 cache, which increases with the instruction footprint. In addition, they neglect the possible presence of idle cores, and do not avoid code and prediction redundancy underutilizing on-chip resources.

Chakraborty *et al.* [2] report high redundancy in instruction fragments across multiple threads executing on multiple cores. They propose CSP, which employs thread migration as a method to distribute the dissimilar fragments of the instructions executed by a thread and group the similar ones together. CSP is limited to separating application level (user) code from OS code, losing opportunities of separation within user code. They point to profile-driven annotation or high-level application directives to identify separation points within user code. TMi generalizes thread migration to include interleaved user-OS code separation points.

Some other recent thread migration proposals target power management, data cache, or memory coherence [13, 11, 16].

6. DISCUSSION

In this section we discuss some of practical implementation aspects and point out directions for future work.

Scalability. Due to limited space, we do not show a range of evaluation setups. We have seen that TMi is scalable on several dimensions: number of cores (4 – 64) and threads (100 – 1K), L1-I cache size (16K – 64K, optimized *FMT*), and instruction footprint sizes (different transaction types).

Workload Dependency. Section 4 shows that TPC-C and TPC-E favor different configuration parameters calling for a dynamic solution. Therefore, as a possible alternative to our algorithm, we might rely on feedback from migration and miss counters.

Impact of OS. Current OS kernels assume full control over thread assignment to cores. To support fast hardware thread migration, thread assignment should be transparent to higher layers, to avoid any software overhead. Otherwise, the OS scheduler, that needs to know where each thread is running, must be informed about these migrations. A different approach might rely on hardware mechanisms to provide counters and migration acceleration, and leave policy choice to software (enabling easier integration with existing schedulers, more flexible policies, etc.).

Thread Identification. A thread migration algorithm that can identify similar threads (TMi-ST) proved more effective than a blind algorithm (TMi-B). In practice, there are two approaches to identify similar threads: (1) relying on the application to transfer this knowledge to the hardware; requires undesirable modifications to the application, and (2) expecting hardware to detect the threads accessing common code segments and tag them as similar threads. We plan to explore these options.

7. CONCLUSIONS

OLTP workloads spend 80% of their time on memory stalls; L1 instruction misses, and L2 data misses. We corroborate these results and show that 91% of L1 capacity misses are for instructions. Additionally, we show that recently proposed replacement policies, which reduce miss rates for some workloads, are ineffective on OLTP workloads. Previous hardware or software proposals are either impractical (require code instrumentation) or relatively expensive (large on-chip data structures).

This work presented a solution based on thread migration, TMi. Similar to CSP [2] and STEPS [6], we exploit the code commonality observed across multiple concurrent threads. Unlike CSP, we do not limit code reuse to OS code segments, and extend that to application code. Unlike STEPS, instead of context switching, we distribute the instruction footprint across multiple cores and migrate execution. We identify the requirements for an ideal system, and implement TMi, an algorithm to evaluate the solution's potential. It is a low-level hardware algorithm which requires no code instrumentation and efficiently utilizes available cache capacity.

TMi was able to reduce the instruction misses by 51% on average. As expected, it impacted the data misses, increasing the total L1-D misses to 95% of the baseline. On an out-of-order pipeline, instruction misses are believed to be more expensive (performance-wise), lending potential to TMi. We believe that data misses could be mitigated by techniques like prefetching, especially when we have a strong clue on what to prefetch.

We identify the requirements for a full practical solution as follows: (a) the ability to identify similar threads, (b) prefetching data for migrating threads, and (c) using accurate cache signatures to locate cache blocks in remote cores. Our next steps will address these requirements and study the timing behavior of TMi.

8. ACKNOWLEDGMENTS

We thank Ioana Burcea and Jason Zebchuk on their invaluable feedback, the members of the AENAO and DIAS laboratories for their support and discussions, Stavros Harizopoulos for his help in the early days of this work, and the reviewers and Ippokratis Pandis for their constructive comments on the paper. This work was partially supported by an NSERC Discovery grant, 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.

9. REFERENCES

[1] I. Burcea, S. Somogyi, A. Moshovos, and B. Falsafi. Predictor virtualization. In *ASPLOS*, 2008.

[2] K. Chakraborty, P. M. Wells, and G. S. Sohi. Computation spreading: employing hardware migration to specialize cmp cores on-the-fly. In *ASPLOS*, 2006.

[3] 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 *ASPLOS*, 2012.

[4] M. Ferdman, C. Kaynak, and B. Falsafi. Proactive Instruction Fetch. In *MICRO*, 2011.

[5] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Temporal instruction fetch streaming. In *MICRO*, 2008.

[6] S. Harizopoulos and A. Ailamaki. Improving instruction cache performance in OLTP. In *TODS*, 2006.

[7] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, Dec. 1989.

[8] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*, 2009.

[9] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA*, 1990.

[10] K. Keeton, D. Patterson, Y. Q. He, R. Raphael, and W. Baker. Performance characterization of a quad pentium pro smp using oltp workloads. In *ISCA*, 1998.

[11] M. Lis, K. S. Shim, M. H. Cho, O. Khan, and S. Devadas. Directoryless shared memory coherence using execution migration. In *PDCS*, 2012.

[12] 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 *PLDI*, 2005.

[13] P. Michaud. Exploiting the cache capacity of a single-chip multi-core processor with execution migration. In *HPCA*, 2004.

[14] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *ISCA*, 2007.

[15] 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 *ASPLOS*, 1998.

[16] K. S. Shim, M. Lis, O. Khan, and S. Devadas. Judicious thread migration when accessing distributed shared caches. In *CAOS*, 2012.

[17] TPC. TPC transcation processing performance council. http://www.tpc.org.