Transactional Prefetching: Narrowing the Window of Contention in Hardware Transactional Memory

Anurag Negi[†] Adrià Armejach[⋆‡] Adrián Cristal[⋆◇] Osman S. Unsal[⋆] Per Stenstrom[†]
[†]Chalmers University of Technology [⋆]Barcelona Supercomputing Center
[‡]Universitat Politècnica de Catalunya [◇]IIIA - CSIC - Spanish National Research Council
{negi, per.stenstrom}@chalmers.se
{adria.armejach, adrian.cristal, osman.unsal}@bsc.es

ABSTRACT

Memory access latency is the primary performance bottleneck in modern computer systems. Prefetching data before it is needed by a processing core allows substantial performance gains by overlapping significant portions of memory latency with useful work. Prior work has investigated this technique and measured potential benefits in a variety of scenarios. However, its use in speeding up Hardware Transactional Memory (HTM) has remained hitherto unexplored. In several HTM designs transactions invalidate speculatively updated cache lines when they abort. Such cache lines tend to have high locality and are likely to be accessed again when the transaction re-executes. Coarse grained transactions that update several cache lines are particularly susceptible to performance degradation even under moderate contention. However, such transactions show strong locality of reference, especially when contention is high. Prefetching cache lines with high locality can, therefore, improve overall concurrency by speeding up transactions and, thereby, narrowing the window of time in which such transactions persist and can cause contention. Such transactions are important since they are likely to form a common TM use-case. We note that traditional prefetch techniques may not be able to track such lines adequately or issue prefetches quickly enough. This paper investigates the use of prefetching in HTMs, proposing a simple design to identify and request prefetch candidates, and measures performance gains to be had for several representative TM workloads.

Categories and Subject Descriptors

C.1 [**Processor Architectures**]: Parallel Architectures

Keywords

hardware transactional memory, multicores, prefetching

PACT'12, September 19–23, 2012, Minneapolis, Minnesota, USA.

1. INTRODUCTION

The ever-widening disparity between the speed at which a processor core can process data and the speed at which the memory hierarchy can supply it has led to a myriad of techniques that aim at overlapping data access latency with some form of useful work. Prefetching is one such technique where, by predicting memory references likely to occur in the near future, data is fetched into structures close to the core before its needed. Various prediction techniques have been employed, targeting frequently encountered patterns in memory references. However, Hardware Transactional Memory (HTM) [13] presents a scenario where a new form of prefetching may be invoked that complements and, in certain scenarios, allows more effective latency hiding than standard techniques.

Several implementations of HTM [9, 6, 25, 19, 18] use first-level caches to isolate speculative state, preserving a consistent state by pushing clean (old) cache lines to second-level caches and beyond. Transactions execute speculatively and any data races detected by the HTM system are typically resolved by forcing one or more of the conflicting transactions to abort. When a transaction aborts speculative state must be discarded and the transaction must be re-executed. To do so, all speculatively modified lines in the first-level cache are invalidated. Subsequent references to such lines during re-execution will miss in the first-level cache and retrieve a clean version of the line from deeper levels of the memory hierarchy. Thus, data transfer latencies delay transactional execution. In scenarios with moderate to high contention this can result in extended transaction execution times, application slow-down and a higher probability of contention. We observe that while a technique like runahead execution [8, 16] could be advantageous here, the hardware requirements for runahead execution and transactional execution are similar (support for checkpointing and dependency tracking) and thus would need to be duplicated in hardware.

In this study we investigate potential gains to be had when lines in the write-set – *the set of speculatively updated cache lines* – of a transaction are prefetched when it begins execution. These lines are highly likely to be referenced again when an aborted transaction re-executes. Interestingly, high contention typically implies high locality of reference. Moreover, in Section 2 we show that this locality of reference is not limited to re-executions of a particular transaction invocation and persists even when a new invocation of the transaction occurs. These observations have motivated the design of hardware prefetching mechanisms described in this paper. These mechanisms are able to track important write-

set lines and are brought into play upon aborts and new transaction starts to prefetch lines that would be required by the transaction during its execution.

The benefits from prefetching write-set lines are expected to be most noticeable in lazy versioning systems like TCC [9, 6]. This is so because, unlike eager versioning designs, they do not restore clean values when speculation fails, and rely upon deeper levels of the memory hierarchy to provide consistent data. However, eager versioning designs like LogTM[26] will benefit from prefetches that are initiated when a new instance of a transaction first begins. In this case a part of the write-set may not be present in the cache when the transaction starts execution, particularly when the contention is high. This effect not only improves execution times but also narrows the window of contention improving concurrency overall.

The rest of this paper is organized as follows. Section 2 shows strong evidence of the locality of reference that exists between multiple invocations of a given transaction for a variety of transactional workloads. This also motivates the hardware structures which are described in detail in Section 3. Section 3 also describes the operation of the prefetch mechanism. Section 4 presents potential performance gains that can be achieved when such prefetching is enabled. We evaluate several transactional prefetching configurations based on the design presented in Section 3, including an idealized variant. Section 5 puts our contributions in perpective of prior work done in prefetching and HTM.

2. MOTIVATION

To make a case for prefetching in transactions we have investigated the behavior of several workloads in the STAMP benchmark suite [5]. The goal of this analysis was to quantify the locality of reference that exists in write-sets across different invocations of the same atomic block or transaction. We recorded all stores issued by each transaction from one thread of each application, tracking the number of transaction invocations that reference each distinct cache line address. We then ranked accessed locations on the basis of frequency of such references for all invocations of each transaction. We choose to concentrate on write-sets for two reasons – first, such lines are likely to get invalidated due to coherence actions or aborts and, second, the read-modify-write nature of common transactions results in a significant overlap with the read-set. The non-overlapping part of the read-set typically sees less contention and is, therefore, likely to be found in the private cache hierarchy.

Figure 1 presents several plots (one for each workload included in the study) that show the number of distinct addresses that can cover a certain fraction of the total number of memory references generated by all invocations of a certain transaction over the duration of the application. For each plot the x-axis is in logarithmic scale and shows the number of distinct addresses, N. The y-axis plots cumulative reference count, C, (for the N most frequently referenced addresses) normalized to the total number of references issued. In other words, if we can track and prefetch a certain number, N, of the most frequently referenced addresses then we can potentially satisfy a fraction, C, of stores in transactions. Moreover, it can be inferred from the read-modify-write behavior of common transactions that these prefetches would also satisfy a significant portion of loads issued by the transaction.

Some transactions have almost no locality of reference, like Tx2 from kernel 1 in SSCA2, a workload with little contention. The linear rise (note that the x-axis is logarithmic) in cumulative reference count is indicative of this fact. A similar case occurs in Labyrinth, where concurrency is limited but the nature of work results in the different invocations of the same transaction updating very different locations. However, for applications like Intruder, Genome, Kmeans and Yada one notices saturation or very low growth in cumulative reference count beyond 16 or 20 addresses, indicating strong locality of reference.

The optimistic nature of TM usually provides good performance when workloads have little contention. However, when contention is high overheads of managing and restoring speculative state grow and increase application execution times. Therefore, to improve HTM design one must aim at minimizing overheads when running applications with moderate to high contention. Besides direct improvements in transaction execution times, prefetching data can potentially improve overall concurrency by narrowing the window of contention for transactions. Figure 2 shows a very simplified view of how this might occur. We view a conflicting access as an event that can occur with equal likelihood at any point during the lifetime of a thread which might intermittently execute transactions. In such a case the probability of contention for the transaction can be represented by the time the thread spends executing the transaction expressed as a fraction of its lifetime. It can be seen that shortening the duration of a transaction reduces the probability of encountering a conflict. Moreover (not shown in the figure) this reduced probability also results in fewer aborts and consequent re-executions (which degrade overall contention even further). This results in fewer conflicting accesses being generated in the system.

Prior work [17] has found that containing transactional stores in dedicated hardware buffers can mitigate overheads associated in reading back lines invalidated on an abort. However, our prefetching technique provides improvements in performance for fresh transaction invocations as well.

As single-thread performance growth stagnates, running applications with inherently limited parallelism in a multi-threaded fashion will be a natural recourse to extract maximum benefit from core count scaling. For such applications in our study (Genome, KMeans, Yada, Intruder) we see that significant locality of reference exists. If we track the 16 most frequently accessed addresses for each transaction we can typically cover more than 60% of the references issued.

3. DESIGN

We subdivide the design into three components – the first which infers locality, the second which manages prefetches and the third which trims prefetch lists. The subsections below describe the structure and behavior of each component. These components are instantiated for each core in a chip-multiprocessor.

3.1 Inferring locality

To decide which cache line addresses are most suitable for prefetching one must first get a measure of the associated locality. The key problem that arises when one attempts to track locality traits of arbitrary memory locations is that

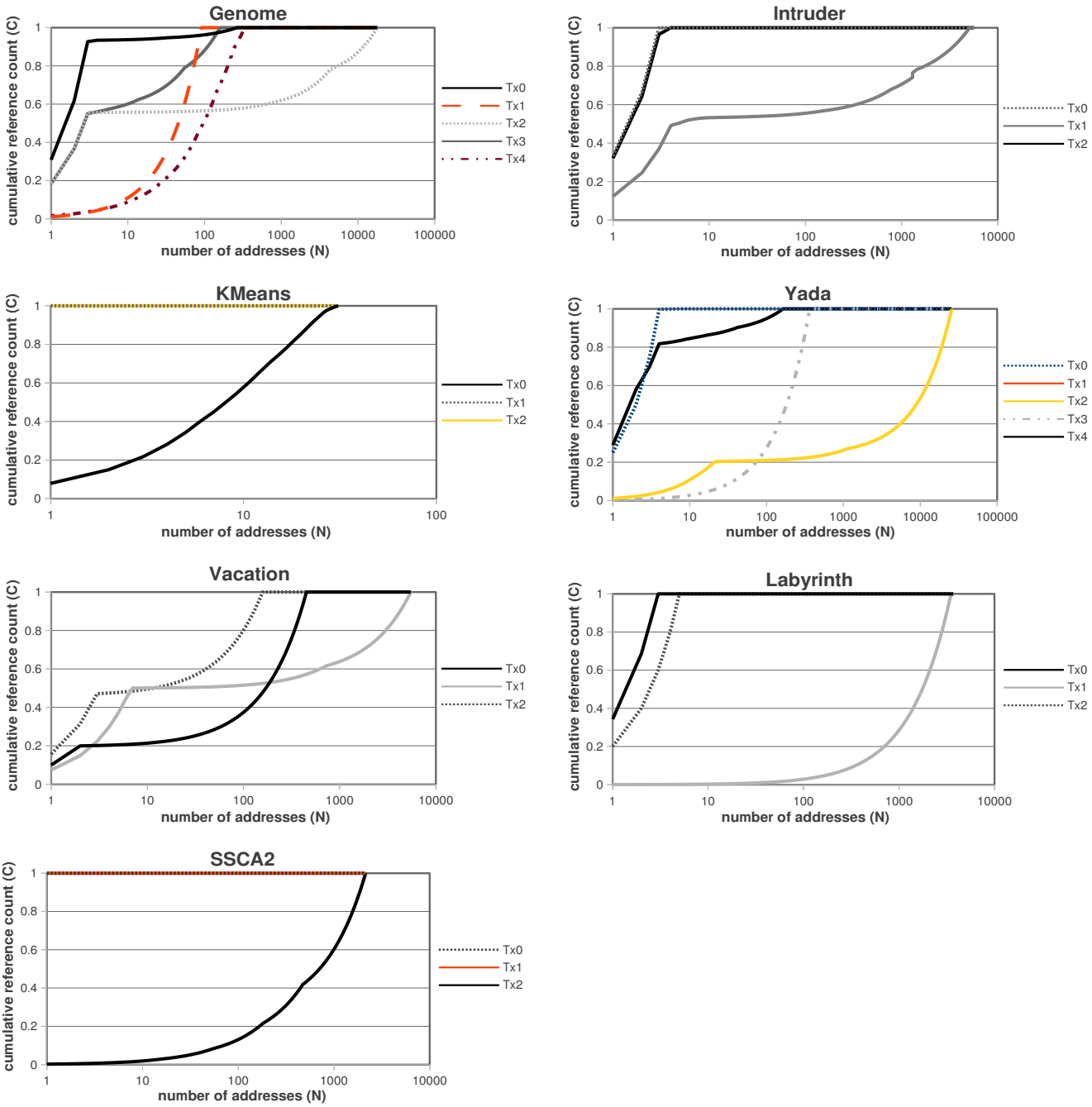

Figure 1: Locality of reference across transaction invocations.

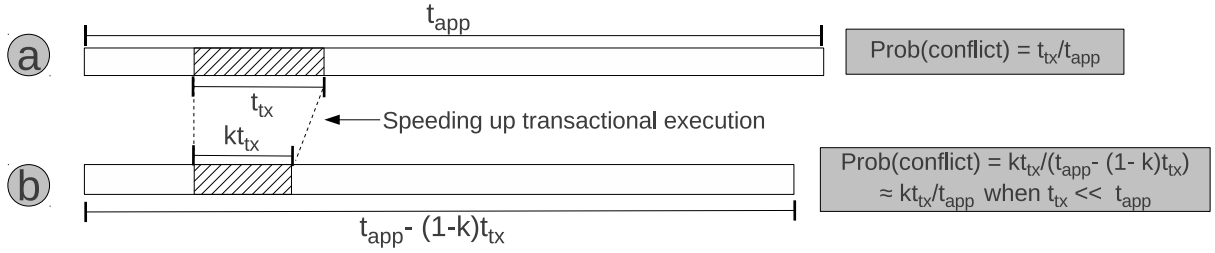

Figure 2: Narrowing the window of contention: effect on conflict probabilities.

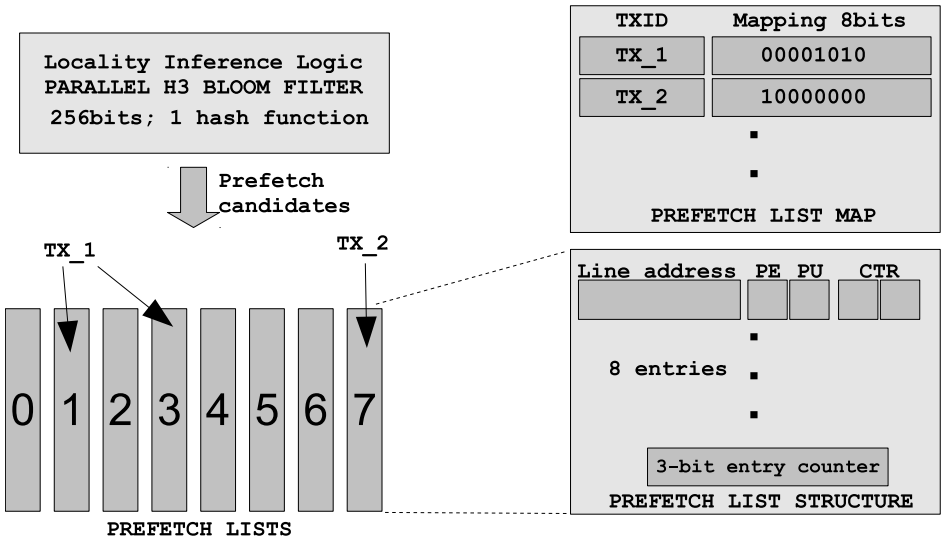

Figure 3: Transactional Prefetching: Key components.

of maintaining a history of memory references until there is enough to infer useful behavioural characteristics. While the history is being recorded there might not be any notion of relative importance of different addresses, resulting in seemingly very large storage requirements or extremely long delays in making inferences. A trade-off must be made that keeps the design simple yet responsive. We choose to do so by employing one or two bloom filters [3] to track memory access history for one or more invocations in the past. Performance evaluations presented later will show performance differences between designs with one and two bloom filters. The single filter design uses a two-step iterative refinement mechansim to learn high-locality cache line addresses one transaction at a time. The two-filter design (described later in Section 4.3) uses a 3-step mechanism using the bloom filters in a ping pong fashion.

When employing a single-bloom filter, two invocations of a transaction are required to learn prefetch candidates. Figure 3 shows the key elements of the proposed mechanism. During the first invocation, cache line addresses in the write set are added to the bloom filter. During the second invocation, cache line addresses targeted by stores are checked for presence in the bloom filter. If a positive match is found the address is added to one of the prefetch candidate lists. Either free lists are used or the lists allocated to the least recently invoked transaction are freed, as explained in Section 3.2.

Locality inference is not initiated for transactions as long as they have prefetch resources allocated to them. Training is aborted if two invocations of another transaction are seen and a watchdog timer has been triggered. This prevents seldom executed transactions from permanently blocking access to locality inference structures. We employ parallel bloom filters employing high-quality H3 hash functions, which have been found, in prior work [22], to be suitable for hardware implementation. The evaluation includes results for both real and perfect bloom filters. Note that the performance (false-positive rate) of these filters is not as critical to performance as when using such filters for conflict detection.

3.1.1 Training in parallel

It is conceivable that the bloom filters can be used to train on more than one transaction simultaneously. This would involve inserting *(address, transaction id)* tuples instead of just addresses into the bloom filters. Since it is non-trivial to selectively delete entries from a bloom filter, such a design must consider the cost of false positives, transaction invocation frequencies and overall responsiveness. We do not study the concept further in this work.

3.2 Managing prefetches

Prefetch candidates produced through locality inference are stored in one or more of several prefetch lists. For the purposes of this study we have 8 lists, 8 entries each. Thus we can support 8 distinct transactions or atomic blocks. If there are fewer transactions which require more than 8 prefetch entries, two or more lists can be chained together. This is managed by the Transactional Prefetch List Map (TPLM), as shown in Figure 3. This is a structure with 8 entries. Each entry contains a TXID (transaction identifier) field and an 8-bit map with high bits indicating prefetch lists allocated to the transaction. Chaining lists together provides flexibility in dealing with transactions of different sizes. When more than 8 transactions exist or no prefetch lists are available we employ an LRU scheme to release resources for the least recently invoked transaction. Prefetches are issued when a transaction begins and has prefetch lists associated with it. In our experiments with lazy conflict resolution designs, it is safe to not regard prefetches as transactional accesses.

Each entry in the prefetch list contains the cache line address, a PE (prefetch enable) bit, a PU (Prefetch Useful) bit and a 2-bit counter. The PE bit is set when the corresponding line is invalidated or evicted from the cache or when a transactional store updates it. Lines with PE bit set to 0 are not prefetched. Transactional commits reset all PE bits. PE bits are also reset when a cache line fill occurs and a transactional update to the line has not yet been issued. All PU bits are reset when a transaction begins. The PU bit

Cores	32 in-order 2GHz Alpha cores, 1 IPC
L1 Caches	32KB 4-way, 64B lines, 1-cycle hit
Bloom filter	256-bit, parallel H3, 1 hash function
L2 Cache	1MB/bank 8-way, 64B lines, 10-cycle hit
Memory	4GB, 150-cycle latency
Interconnect	2D mesh, 2 cycles per hop
Directory	full-bit vector sharers list, 10-cycle directory latency

Table 1: Simulation parameters.

Benchmark	Input parameters
Genome	-g4096 -s128 -n524288
Intruder	-a10 -l32 -n8192 -s1
KMeans	-m15 -n15 -t0.05 -i n32768-d24-c16
Labyrinth	-i random-x96-y96-z3-n384.txt
SSCA2	-s13 -i1.0 -u1.0 -l3 -p3
Vacation	-n8 -q40 -u90 -r1048576 -t32768
Yada	-a20 -i ttimeu10000.2

Table 2: Evaluated STAMP benchmarks and input parameters.

is set when a transactional store targets the corresponding cache line, indicating that the address still retains locality.

3.3 Trimming prefetch lists and transactions

The two bit counter for each prefetch candidate is set to 4 when the entry is first created. On transaction commits the counter is decremented for all entries in the prefetch list for which PU bit is not set. If this counter reaches 0 the line is not prefetched any more. If all entries for a certain transaction have counts set to 0, the resources (prefetch lists and TPLM) are released for use by other transactions. This is easily achieved by associating a 3-bit counter with each prefetch list. It tracks the number of active prefetch candidates in the list. It is incremented when entries are added to the prefetch list after training and decremented every time a prefetch candidate is trimmed. When the counter is decremented to zero the corresponding bit in the list allocation bit-map in the TPLM is reset. When all bits in the list allocation bit map of a transaction are set to zero, the entry can be reused. The next invocation of such a trimmed transaction will be eligible for locality inference, when prefetch lists will be rebuilt.

4. EVALUATION

In this section we evaluate the performance of transactional prefetching. We use as baseline Scalable-TCC, a state-of-the-art lazy HTM system. We first describe the simulation environment that we use, then we present our preliminary results.

4.1 Simulation Environment

For the evaluation we use M5 [2], an Alpha 21264 full-system simulator. We modify M5 to faithfully model the Scalable-TCC proposal to operate in a chip multi-processor (CMP) with private L1 caches and a banked L2 cache. The design is configured to avoid rare overflows of transactional data from private caches. These are handled using a special overflow buffer. In our experiments only a few such events are noticed. Scalable-TCC has an always-in-transaction approach and employs lazy conflict detection and resolution at commit time, transactional updates are kept in private buffers (caches) to maintain isolation. Note that the prefetch mechanism is invoked only for transactions defined in the application source code. Table 1 summarizes the system parameters that we use, with one level of private cache and a 2D mesh network connecting the shared L2 banks, resembling the Scalable-TCC proposal. Our proposed transactional prefetching scheme is implemented on top of the baseline HTM. This detailed simulation model , denoted as TP (Transactional Prefetching), employs one 256-bit H3 bloom filter. In addition, we also simulate an idealized model that at the beginning of a transaction prefetches all the lines that have been speculatively written by that transaction in the past. These prefetches are considered to be serviced instantaneously. We name this model PA (Prefetch All).

We use the STAMP benchmark suite [5] to evaluate our proposal. Table 2 lists the evaluated workloads and input parameters. We exclude the application Bayes from our evaluation, because this application has non-deterministic exiting conditions leading to severe load imbalance between threads, which makes comparison between different systems inconclusive.

4.2 Performance Results

Figure 4 shows the execution time breakdown for the HTM systems that we evaluate, namely Scalable-TCC (S), Transactional Prefetching (TP), and Prefetch All (PA). The results in Figure 4 are normalized to Scalable-TCC 32-threaded executions, and they are split into six parts, namely Barrier, Commit, Useful, StallCache, Wasted, and WastedCache. The component, Useful, is defined as one cycle per instruction plus the number of memory accesses per instruction multiplied by the L1D hit latency; the component, StallCache, is defined as the time spent waiting for an L1D cache miss to be served. Analogously, for aborted transactions we define Wasted and WastedCache.

Intruder shows remarkable improvement when prefetching is enabled (a speedup of more than 30%). It is a highly contended application exhibiting significant locality across various transaction invocations. In this scenario prefetching data results in substantial shortening of transaction lifetimes. The components, StallCache and WastedCache, show major reductions, as can be seen in Figure 4. We highlight this application because in our opinion it is an important workload that is representative of applications with limited concurrency. Such multithreaded applications will gain importance as parallelization is expected to become the only source of performance scaling.

Genome shows moderate contention and a significant amount of locality for most transactions (see Figure 1). It shows two distinct phases during execution – a short early high contention phase followed by a longer phase with low to moderate contention. The benefits of prefetching accrue in the first phase, yielding an 16% improvement over the baseline.

Yada is another application with moderate contention (see Figure 1). Prefetching lines improves performance, though not by much (3%). One of the reasons for this is limited tracking resources at the prefetcher. Yada has large transactions, and the number of prefetched addresses constitutes a small fraction of the memory references generated.

Vacation has large transactions, there is very little con-

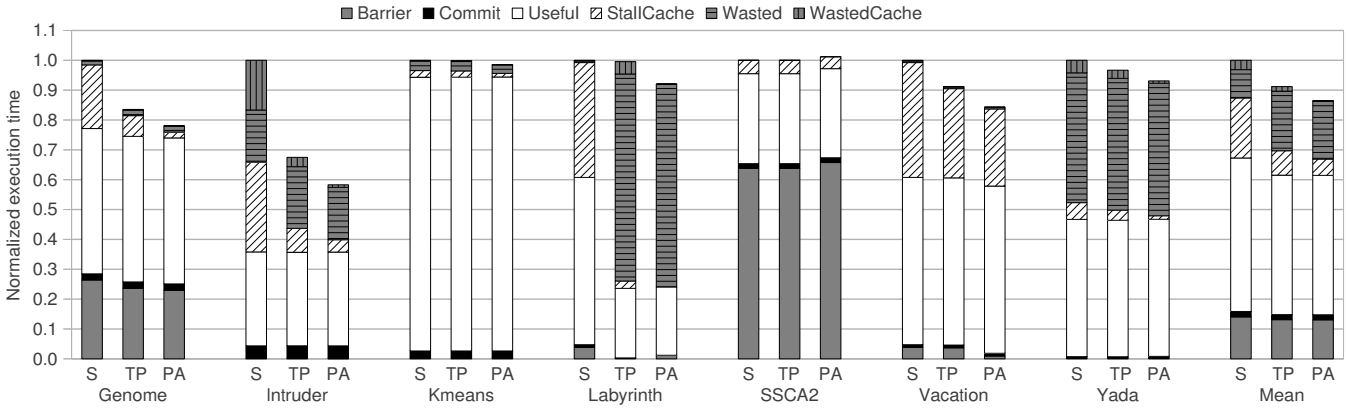

Figure 4: Normalized execution time breakdown for 32 threads.
S — Scalable-TCC; **TP** — Transactional Prefetching; **PA** — Prefetch All

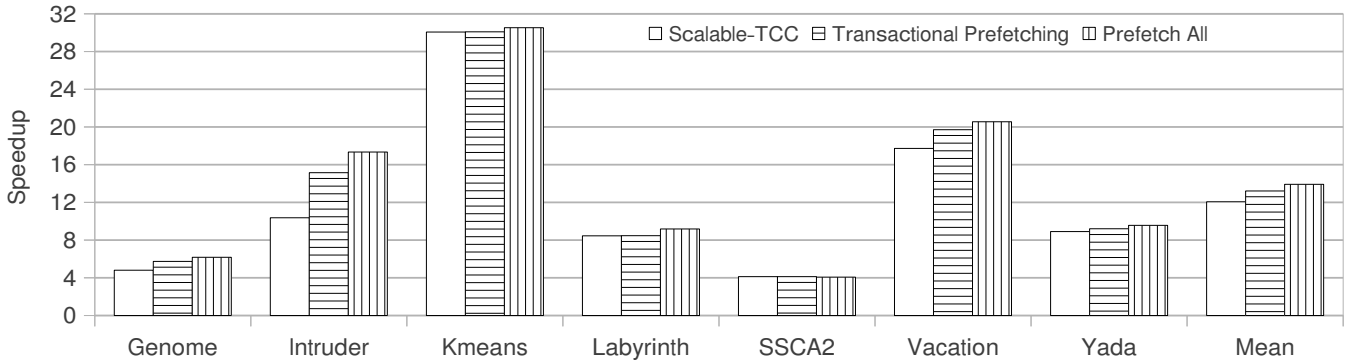

Figure 5: Scalability for 32 threaded workloads.

Benchmark	%Useful	%Trimmed	%Cache improv.	Prefetches per Commit
Genome	92.88	81.36	60.07	0.02
Intruder	99.87	11.11	64.55	4.28
KMeans	41.64	99.54	1.55	0.2
Labyrinth	100.00	0.00	82.74	2.27
SSCA2	0.00	99.49	0.00	0
Vacation	98.04	14.66	14.74	0.03
Yada	98.97	38.64	36.66	2.26
Mean	88.56	40.88	37.19	1.3

Table 3: Statistics of Transactional Prefetching for evaluated workloads.

Legend: **%Useful** — Percentage of useful prefetches compared to issued prefetches; **%Trimmed** — Percentage of trimmed entries compared to added in prefetch lists; **%Cache improvement** — Percentage improvement of total cache service time compared to Scalable-TCC; **Prefetches per commit** – Average number of prefetches issued per committed transaction.

tention and transactions are read dominant. The dominant transaction (shown as Tx1 in Figure 1) has good locality of reference and the TP configuration is able to take advantage of this, yielding an improvement of about 15% over the baseline (as seen in Figure 6).

KMeans exhibits short phases with some degree of locality. This is evident from the number of trimmed and useful prefetches as shown in Table 3. However, transactions do not appear to have a dominant effect on execution time in this application. We, therefore, do not notice any appreciable deviation in execution times across various configurations.

SSCA2 is a highly concurrent application with little contention and almost no locality across transactions(see Figure 1). Hence, prefetching is not expected to play a role here, and as shown in Table 3 our proposed prefetch mechanism issues just 35 prefetches spread over more than 100,000 transaction invocations. Moreover, these prefetches get trimmed from the lists rapidly (as indicated by the high (99.49%) percentage of trimmed prefetches, see Table 3). Though

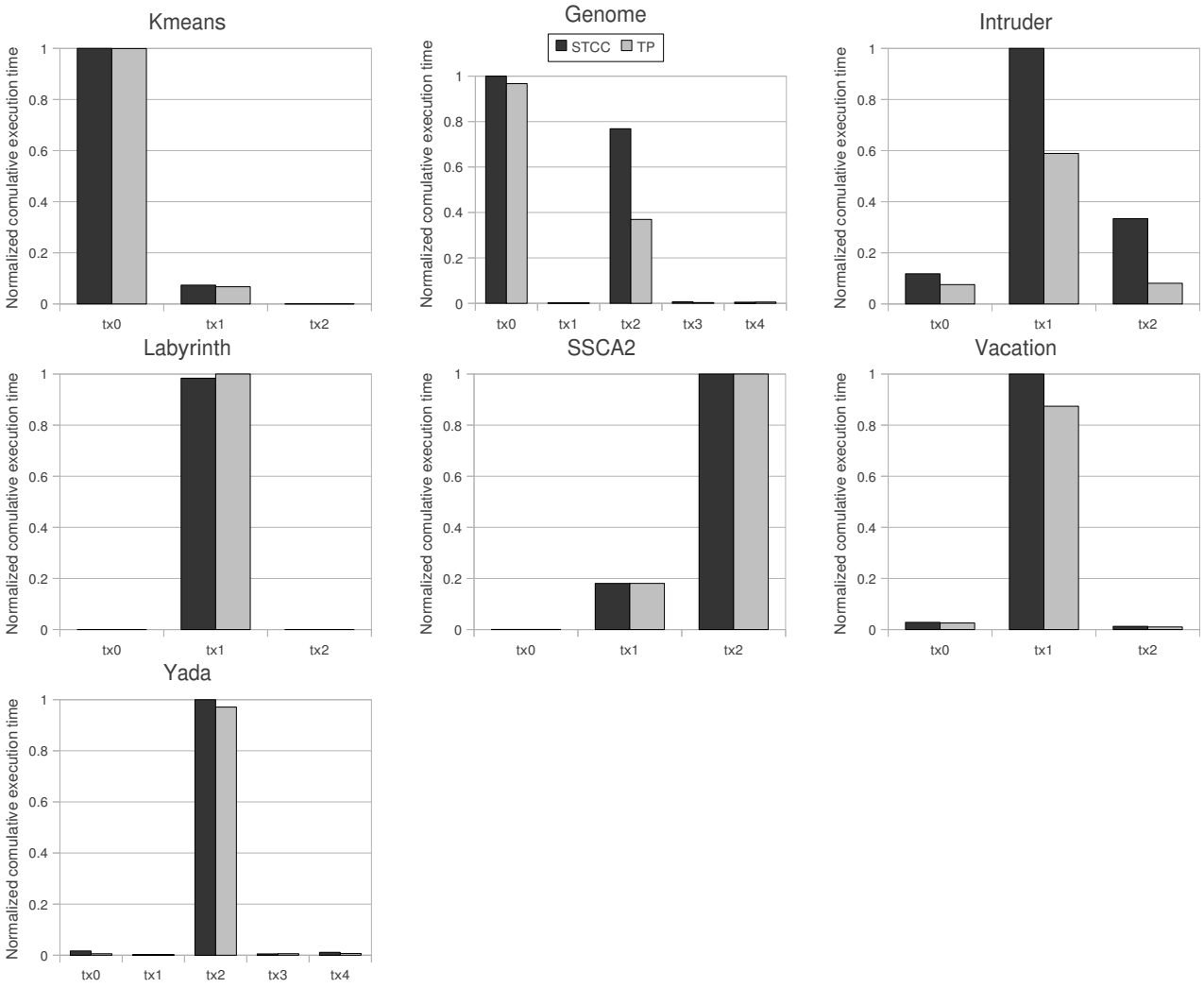

Figure 6: Impact on transaction execution times

Labyrinth repeatedly accesses a large set of addresses, it executes a very small number of transactions (less than ten instances of each defined transaction), leading to negligible performance gains for the TP configuration. Moreover, due to Labyrinth's lack of parallelism and sensitivity to transaction interleaving, some configurations exhibit increased contention and therefore more wasted work, as can be seen in Figure 4.

Figure 5 shows the scalability chart for the evaluated workloads using 32 threads on 32 cores. Intruder has a remarkable boost in scalability reaching 15.3× with TP, a promising result for an application that is known to have difficulties to scale. Noticeable improvements can be also seen in Genome, KMeans, Vacation and Yada, while applications like SSCA2 and Labyrinth remain flat due to their transactional characteristics.

Overall, as shown in Table 3 our transactional prefetching mechanism successfully infers locality from the evaluated workloads, achieving more than 90% utilization of issued prefetches for all applications except KMeans, where locality is high, but appears in short phases. Moreover, our design is able to detect scenarios where prefetching is not useful, for example in applications like SSCA2, and does not issue useless prefetches for such scenarios. In general, as can be observed in Table 3, if the usefulness of prefetches is low then the number of issued prefetches per committed transaction is rather small as well.

Figure 6 shows relative cumulative execution times for each transaction defined in code. Only successful commits have been considered. From these numbers it is possible to estimate the impact of transactional prefetching. For each transaction, two bars are shown – the left one corresponds to execution time seen with the baseline design and the right one corresponds to execution time with prefetching enabled. It is instructive to compare these numbers to those shown in Figure 1. We can see that applications which show high locality (Genome, Intruder, Yada, Vacation) also see an improvement in execution times. The improvement moreover is proportional to the degree of locality seen – for example, in Intruder most transactional accesses target only a few addresses and hence, we see a larger improvement than that seen in Yada. SSCA2 and Kmeans do not show much improvement since there is little locality.

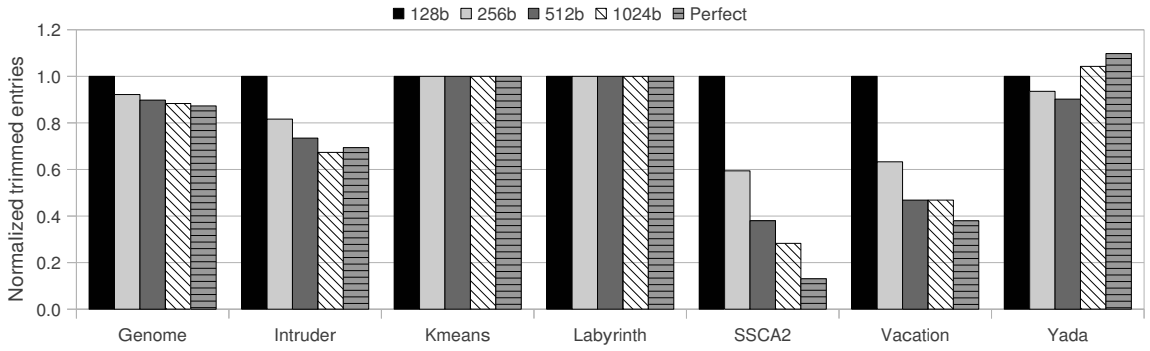

Figure 8: Impact of bloom filter size on trimmed entries.

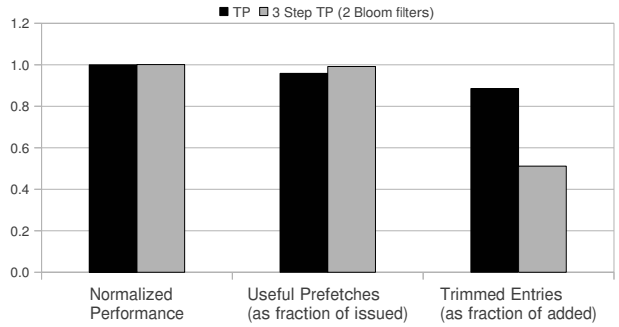

Figure 7: Impact of the 2-filter 3-step approach.

4.3 Two Filter Approach

More accurate learning of prefetch candidates can be performed by employing two bloom filters in a ping-pong fashion. We test this approach by using a 3-step interative refinement design. The first step inserts write-set cache line addresses in one (BFx) of the two filters (BFx and BFy). The next invocation of the transaction triggers the second step wherein written lines that are found in BFx are inserted in BFy. The third invocation starts the final step of the learning process, filling prefetch candidate lists based on written lines that are found in BFy. As before, we train one transaction at a time, releasing training resources on completion. This approach enables more accurate learning at the cost of responsiveness (it takes longer to train). Figure 7 shows how this approach compares against the single filter approach in terms of overall performance and the fraction of useful and trimmed prefetches. Although there is no appreciable difference in performance the two filter approach generates more accurate prefetches, as indicated by a larger fraction of useful prefetches and fewer trims.

4.4 Sensitivity Analysis

Bloom filter configuration has little impact of performance of transactional workloads. We varied bloom filter sizes, ranging from 128 bits to 1024 bits and also implemented perfect signatures. There is remarkable consistency in execution times across different filter configurations. This is because few additional prefetches arising from increased false positives with small bloom filter sizes have negligible impact on performance and are quickly trimmed from prefetch lists. Figure 8 shows that smaller filters result in more trimmed entries. However, in the case of Yada, variations in behavior induced by transaction interleaving cause minor deviation in the number of trimmed entries (with a 2% spread in execution times).

5. RELATED WORK

Although the first proposal by Herlihy and Moss [11] appeared in 1993, research in TM gained momentum with the introduction of multicore architectures. Two early HTM proposals, TCC [10] and LogTM [26], explore two very different points in the HTM design space. TCC defines a lazy conflict resolution design where transactions execute speculatively until one tries to commit its results and causes the re-execution of any concurrent conflicting transaction. LogTM describes an eager conflict resolution design that employs coherence to detect conflicts as soon as they occur and are resolved by asking the requester to retry (with a way to break occasional deadlocks through software intervention). Since then a lot of work has been done targeting a host of different issues that arise when transactional applications run on multicores. Bobba et al. [4] categorized pathologies that can arise in fixed policy HTM designs and degrade scalability and performance. The paper pointed out performance bottlenecks that can arise out of limited commit bandwidth in lazy conflict resolution designs and overheads due to excessive aborts in eager resolution designs. Several designs since then have targeted improved scalability in lazy conflict resolution systems through various means – making write-set commits more fine-grained [6, 20, 21] and ensuring conflicting transactions do not interfere with an on-going commit [25, 18]. Others have attempted to reduce abort overheads in both eager and lazy conflict resolution systems – by allowing eager systems to utilize deeper levels of the memory hierarchy to buffer old values [14] and by having caches with special SRAM cells that can store two versions of the same line simultaneously [1]. Yet others have attempted to incorporate the best of both eager and lazy policies in one design – at the granularity of application phases [19], at the granularity of transactions [15] , and at the granularity of cache lines [24]. There exist studies that have attempted to insulate the coherent cache hierarchy from adverse effects of repeated aborts [17]. These varied attempts at reducing overheads involved in shared data accesses by cooperating threads have motivated the design effort in this work. This paper, however, presents a study and design that is largely orthogonal to the various design approaches discussed above. It uses the fact that transactions show locality of reference which can be utilized to improve the speed at which they can

complete updates to shared data, thereby improving speed and reducing contention.

Several prior studies have developed ideas regarding cache line prefetching [12, 23] and investigated various prefetching schemes based on detecting cache-miss patterns in non-transactional workloads. This paper, unlike prior work, describes a scheme that does not rely upon the existence of a simple pattern (like a stride) in the memory reference stream. It can learn arbitrary sets of cache line addresses as long as they show locality of reference across multiple invocations of the same section of code. Thus, this proposed technique is expected to be complementary to others. Moreover, with this technique prefetches can be issued earlier than in other techniques. Chou et. al [7] present epoch-based correlation prefetches which utilize special hardware and software support structures to detect prefetch trigger events and manage prefetch candidates. Our work presents a simpler, less expensive interface to manage and trigger prefetches using low complexity per-core hardware.

6. CONCLUSIONS AND FUTURE WORK

This paper highlights the importance of prefetching data in the new context of hardware transactional memory. Since transactions are used to annotate parts of multithreaded algorithms where concurrent tasks share information, it is important that they run as fast as possible to improve overall scalability of the application. Moreover transactions are clearly demarcated sections of code and thus can be targeted by techniques, such as the one proposed in this paper, that attempt to utilize any locality of reference that may exist within such codes. Our technique, using relatively modest hardware support shows improvements for most transactional workloads we have analyzed, with substantial gains of upto 35% under high contention (for intruder).

In the future we would like to enhance this technique and apply it to other scenarios to accelerate generic blocks of code that exhibit high locality of reference across invocations. We feel that critical sections and synchronization operations could also benefit from such prefetching. The observation that high contention is indicative of high locality makes this technique potentially advantageous in mitigating the impact of data-sharing bottlenecks in multithreaded applications. We also wish to study interactions when this technique is combined with other forms of prefetching, using the insights so acquired to develop synergistic techniques that further improve the design to speed up both transactional and non-transactional code.

Acknowledgements

This paper is the result of a research collaboration between Chalmers University of Technology and Barcelona Supercomputing Center (BSC) made possible through a collaboration grant to Adria Armejach by the European HiPEAC Network of Excellence. Anurag Negi's work at Chalmers is supported by grants from the Swedish Foundation for Strategic Research (SSF) under the SCHEME project. Adrià's work is supported by the agreement between the Barcelona Supercomputing Center and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union under contracts TIN2007-60625 and TIN2008-02055-E.

7. REFERENCES

[1] A. Armejach, A. Seyedi, R. Titos-Gil, I. Hur, O. S. Unsal, A. Cristal, and M. Valero. Using a reconfigurable L1 data cache for efficient version management in hardware transactional memory. In *PACT '11: Proc. 20th International Conference on Parallel Architectures and Compilation Techniques*, 2011.

[2] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi, and S. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 2006.

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, July 1970.

[4] J. Bobba, K. E. Moore, L. Yen, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[5] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of The IEEE International Symposium on Workload Characterization*, Sep 2008.

[6] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, 2007.

[7] Y. Chou. Low-Cost Epoch-Based Correlation Prefetching for Commercial Applications. In *40th Annual IEEE/ACM International Symposium on Microarchitecture, 2007*, pages 301 –313, dec. 2007.

[8] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *In Proceedings of the 1997 International Conference on Supercomputing*, pages 68–75, 1997.

[9] L. Hammond, B. D. Carlstrom, V. Wong, M. Chen, C. Kozyrakis, and K. Olukotun. Transactional coherence and consistency: Simplifying parallel hardware and software. *IEEE Micro*, 24(6), Nov-Dec 2004.

[10] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st International Symposium on Computer Architecture*, June 2004.

[11] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[12] 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*, ISCA '90, pages 364–373, 1990.

[13] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[14] M. Lupon, G. Magklis, and A. Gonzalez. FASTM: A log-based hardware transactional memory with fast abort recovery. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2009.

[15] M. Lupon, G. Magklis, and A. González. A dynamically adaptable hardware transactional

memory. In *In Proc. of 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2010.

[16] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *In HPCA-9*, pages 129–140, 2003.

[17] A. Negi, R. Titos-Gil, M. E. Acacio, J. M. Garcia, and P. Stenstrom. Eager Meets Lazy: The impact of write-buffering on hardware transactional memory. *International Conference on Parallel Processing (ICPP)*, pages 73–82, 2011.

[18] A. Negi, R. Titos-Gil, M. E. Acacio, J. M. Garcia, and P. Stenstrom. π-TM: Pessimistic Invalidation for Scalable Lazy Hardware Transactional Memory (poster). In *Parallel Architectures and Compilation Techniques (PACT) 2011*, Oct. 2011.

[19] A. Negi, M. Waliullah, and P. Stenstrom. LV*: A low complexity lazy versioning HTM infrastructure. In *Proc. of the Intl. Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS 2010)*, pages 231–240, July 2010.

[20] S. H. Pugsley, M. Awasthi, N. Madan, N. Muralimanohar, and R. Balasubramonian. Scalable and reliable communication for hardware transactional memory. In *PACT '08: Proc. 17th international conference on Parallel architectures and compilation techniques*, pages 144–154, Oct. 2008.

[21] X. Qian, W. Ahn, and J. Torrellas. Scalablebulk: Scalable cache coherence for atomic blocks in a lazy environment. In *In Proc. of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2010*, pages 447–458, 2010.

[22] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing signatures for transactional memory. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[23] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 33, pages 42–53, 2000.

[24] R. Titos-Gil, A. Negi, M. E. Acacio, J. M. García, and P. Stenstrom. ZEBRA: a data-centric, hybrid-policy hardware transactional memory design. In *Proceedings of the international conference on Supercomputing*, ICS '11, 2011.

[25] S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: eager-lazy hardware transactional memory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2009.

[26] L. Yen, J. Bobba, M. M. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, Feb. 2007.