Three hours

Two academic papers are provided for use with the examination.

UNIVERSITY OF MANCHESTER
SCHOOL OF COMPUTER SCIENCE

Designing for Parallelism and Future Multi-core Computing

Date: Monday 12th January 2015
Time: 09:45 - 12:45

Please answer the single Question provided

Two academic papers are attached for use with the examination.
Otherwise this is a CLOSED book examination.
The use of electronic calculators is NOT permitted.
1. Provide an analysis of one of the following two papers:
   A) SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding
      HPCA 2012
   or
   B) Virtues and Limitations of Commodity Hardware Transactional Memory
      PACT 2014

by answering the following questions:

a) What is the problem being addressed? (10 marks)
b) What is the proposed solution? (12 marks)
c) What are the assumptions? (6 marks)
d) How is it evaluated? (12 marks)
e) What are the limitations? (6 marks)
f) Overall assessment of paper and possible improvements? (4 marks)

(Total 50)
SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding

Daniel Sanchez and Christos Kozyrakis
Stanford University
\{sanchezd, kozyraki\}@stanford.edu

Abstract

Large-scale CMPs with hundreds of cores require a directory-based protocol to maintain cache coherence. However, previously proposed coherence directories are hard to scale beyond tens of cores, requiring either excessive area or energy, complex hierarchical protocols, or inexact representations of sharer sets that increase coherence traffic and degrade performance.

We present SCD, a scalable coherence directory that relies on efficient highly-associative caches (such as zcaches) to implement a single-level directory that scales to thousands of cores, tracks sharer sets exactly, and incurs negligible directory-induced invalidations. SCD scales because, unlike conventional directories, it uses a variable number of directory tags to represent sharer sets: lines with one or few sharers use a single tag, while widely shared lines use additional tags, so tags remain small as the system scales up. We show that, thanks to the efficient highly-associative array it relies on, SCD can be fully characterized using analytical models, and can be sized to guarantee a negligible number of evictions independently of the workload.

We evaluate SCD using simulations of a 1024-core CMP. For the same level of coverage, we find that SCD is 13\times more area-efficient than full-map sparse directories, and 2\times more area-efficient and faster than hierarchical directories, while requiring a simpler protocol. Furthermore, we show that SCD’s analytical models are accurate in practice.

1. Introduction

As Moore’s Law enables chip-multiprocessors (CMPs) with hundreds of cores [15, 28, 30], implementing coherent cache hierarchies becomes increasingly difficult. Snooping cache coherence protocols work well in small-scale systems, but do not scale beyond a handful of cores due to their large bandwidth overheads, even with optimizations like snoop filters [18]. Large-scale CMPs require a directory-based protocol, which introduces a coherence directory between the private and shared cache levels to track and control which caches share a line and serve as an ordering point for concurrent requests. However, while directory-based protocols scale to hundreds of cores and beyond, implementing directories that can track hundreds of sharers efficiently has been problematic. Prior work on thousand-core CMPs shows that hardware cache coherence is important at that scale [18, 20], and thousand-core directory-coherent CMPs are already on the market [30], stressing the need for scalable directories.

Ideally, a directory should satisfy three basic requirements. First, it should maintain sharer information while imposing small area, energy and latency overheads that scale well with the number of cores. Second, it should represent sharer information accurately — it is possible to improve directory efficiency by allowing inexact sharer information, but this causes additional traffic and complicates the coherence protocol. Third, it should introduce a negligible amount of directory-induced invalidations (those due to limited directory capacity or associativity), as they can significantly degrade performance.

Proposed directory organizations make different trade-offs in meeting these properties, but no scheme satisfies all of them. Traditional schemes scale poorly with core count: Duplicate-tag directories [2, 29] maintain a copy of all tags in the tracked caches. They incur reasonable area overheads and do not produce directory-induced invalidations, but their highly-associative lookups make them very energy-inefficient with a large number of cores. Sparse directories [13] are associative, address-indexed arrays, where each entry encodes the set of sharers, typically using a bit-vector. However, sharer bit-vectors grow linearly with the number of cores, making them area-inefficient in large systems, and their limited size and associativity can produce significant directory-induced invalidations. For this reason, set-associative directories tend to be significantly oversized [10]. There are two main alternatives to improve sparse directory scalability. Hierarchical directories [31, 33] implement multiple levels of sparse directories, with each level tracking the lower-level sharers. This way, area and energy grow logarithmically with the number of cores. However, hierarchical organizations impose additional lookups on the critical path, hurting latency, and more importantly, require a more complex hierarchical coherence protocol [31]. Alternatively, many techniques have been explored to represent sharer sets inexactly through coarse-grain bit-vectors [13], limited pointers [1, 6], Tagless directories [35] and SPACE [36]. Unfortunately, these methods introduce additional traffic in the form of spurious invalidations, and often increase coherence protocol complexity [35].

In this paper, we present the Scalable Coherence Directory (SCD), a novel directory scheme that scales to thousands of cores efficiently, while incurring negligible invalidations and keeping an exact sharer representation. We leverage recent prior work on efficient highly-associative caches (ZCache [25] and Cuckoo Directory [10]), which, due to their multiple hash functions and replacement process, work in
First, we recognize that, to be scalable, a directory implementation only needs the number of bits per tracked sharer to scale gracefully (e.g., remaining constant or increasing logarithmically) with the number of cores. SCD exploits this insight by using a variable-size sharer set representation: lines with one or few sharers use a single directory tag, while widely shared lines use additional tags. We propose a hybrid pointer/bit-vector organization that scales logarithmically and can track tens of thousands of cores efficiently. While conventional set-associative arrays have difficulties with this approach, highly-associative arrays allow SCD to work.

We develop analytical models that characterize SCD and show how to size it. First, we show that for a given occupancy (fraction of directory capacity used), SCD incurs the same number of directory-induced invalidations and average number of lookups, independently of the workload. Second, different workloads impose varying capacity requirements, but the worst-case capacity requirement is bounded and small. Hence, directories can be built to guarantee a negligible number of invalidations and a small average number of lookups in all cases, guaranteeing performance and energy efficiency with just a small amount of overprovisioning (around 5-10% depending on the requirements, much smaller than what is required with set-associative arrays [10]). These results are useful for two reasons. First, they enable designers to quickly size directories without relying on empirical results and extensive simulations. Second, they provide guarantees on the interference introduced by the shared directory, which is paramount to achieve performance isolation among multiple competing applications sharing the chip (CMPs need a fully shared directory even if they have private last-level caches). This analytical characterization also applies to sparse directories implemented with highly-associative caches (Cuckoo Directories), which prior work has studied empirically [10].

We evaluate SCD by simulating CMPs with 1024 cores and a 3-level cache hierarchy. We show that, for the same level of provisioning, SCD is $13 \times$ more area-efficient than sparse directories and $2 \times$ more area-efficient than hierarchical organizations. SCD can track 128MB of private cache space with a 20MB directory, taking only 3% of total die area. Moreover, we show that the analytical models on invalidations and energy are accurate in practice, enabling designers to guarantee bounds on performance, performance isolation, and energy efficiency.

2. Background and Related Work

In this section, we provide the necessary background for SCD, reviewing previously proposed directory organizations and efficient highly-associative arrays, which SCD relies on.

2.1. Directory Organizations

Cache coherence is needed to maintain the illusion of a single shared memory on a system with multiple private caches. A coherence protocol arbitrates communication between the private caches and the next level in the memory hierarchy, typically a shared cache (e.g., in a CMP with per-core L2s and a shared last-level cache) or main memory (e.g., in multi-socket systems with per-die private last-level caches). In this work we focus on directory-based, write-invalidate protocols, as alternative protocols scale poorly beyond a few private caches [14]. These protocols use a coherence directory to track which caches share a line, enforce serialization by invalidating or downgrading access permissions for sharers, and act as an ordering point for concurrent requests to the same address. Implementing a directory structure that scales to hundreds of sharers efficiently has been problematic so far. We now review different directory organizations, with a focus on comparing their scalability. Table 1 summarizes their characteristics.

Traditional directory schemes do not scale well with core count. Duplicate-tag directories maintain a full copy of all the tags tracked in the lower level. Their area requirements scale well with core count, but they have huge associativity requirements (e.g., tracking 1024 16-way caches would require 16384 ways), so they are limited to small-scale systems [2, 29]. In contrast, sparse directories [13] are organized as an associative array indexed by line address, and each directory tag encodes the set of sharers of a specific address. Sparse directories are energy-efficient. However, due to their limited associativity, sparse directories are often forced to evict entries, causing directory-induced invalidations in the lower levels of the hierarchy. This can pose large performance overheads and avoiding it requires directories that are significantly overprovisioned [10].

The encoding method for the sharer set is a fundamental design choice in sparse directories. Full-map sparse directories encode the sharer set exactly in a bit-vector [13]. They support all sharing patterns, but require storage proportional to the number of cores, and scale poorly beyond a few tens of cores. Alternatively, sparse directories can use a compressed but inexact encoding of the sharer set. Traditional alternatives include coarse-grain sharer bit-vectors [13], and limited pointer schemes, in which each entry can hold a small number of sharer pointers, and lines requiring more sharers either cause one of the existing sharers to be invalidated, a broadcast on future invalidations and downgrades [1, 20], or trigger an interrupt and are handled by software [6]. Inexact sharer set schemes trade off space efficiency for additional coherence traffic and protocol complexity.

In contrast to these techniques, hierarchical sparse directories [12, 31, 33] allow an exact and area-efficient representation of sharer sets. Hierarchical directories are organized in multiple levels: each first-level directory encodes the sharers of a subset of caches, and each successive level tracks directories of the previous level. A two-level organization can scale to thousands of cores efficiently. For example, using 32 first-level directories and one (possibly banked) second-level directory, we can track 1024 caches using 32-bit sharer vectors, or 4096 caches using 64-bit vectors. However, hierarchical directories have two major drawbacks. First, they require several lookups on the critical path, increasing directory latency and hurting performance. Second, multi-level
coherence protocols are more complex than single-level protocols, and significantly harder to verify [7, 8].

Motivated by the shortcomings of traditional approaches, recent work has investigated alternative directory organizations that improve scalability in a single-level directory. Tagless directories [35] use Bloom filters to represent sharer sets. Tagless does not store address tags, making it highly area-efficient (as we will see later, SCD spends more space storing addresses than actual coherence information). Although Tagless reduces area overheads, both area and energy scale linearly with core count, so Tagless is area-efficient but not energy-efficient at 1024 cores [10]. Additionally, it requires significant modifications to the coherence protocol, and incurs additional bandwidth overheads due to false positives. Moreover, Tagless relies on the tracked caches being set-associative, and would not work with other array designs, such as skew-associative caches [27] or zcaches [25].

Table 1: Qualitative comparison of directory schemes. The first three properties are desirable, while the last three are undesirable.

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Scalable area</th>
<th>Scalable energy</th>
<th>Exact sharers</th>
<th>Dir-induced invalidations</th>
<th>Extra protocol complexity</th>
<th>Extra latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duplicate-tag</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Sparse full-map</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Coarse-grain/lmtr</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Small</td>
<td>No</td>
</tr>
<tr>
<td>Hierarchical sparse</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>High</td>
<td>Yes</td>
</tr>
<tr>
<td>Tagless</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Medium</td>
<td>No</td>
</tr>
<tr>
<td>SPACE</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Small</td>
<td>No</td>
</tr>
<tr>
<td>Cuckoo Directory</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Negligible</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SCD</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Negligible</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

2.2. Efficient Highly-Associative Caches

Recent work has proposed cache designs that provide high associativity with a small number of ways. Both ZCache [25] and Cuckoo Directory [10] build on skew-associative caches [27] and Cuckoo hashing [24]. As shown in Figure 1, these designs use a different hash function to index each way, like skew-associative caches. Hits require a single lookup, but on a replacement, these designs leverage the multiple hash functions to provide an arbitrarily large number of replacement candidates in multiple steps, increasing associativity. Both schemes differ in how the replacement process is implemented.

Cuckoo Directory uses W-ary Cuckoo hashing to implement its replacement process: when inserting a new line, if there are no unused entries, the incoming line replaces one of the existing ones, which is taken out and then reinserted in one of the other positions it can map to. This process is repeated until either an unused entry is found, or a threshold of reinsertion attempts is reached. In this case, the line that was last taken out is evicted. Ferdman et al. build sparse directories using this technique and empirically show that it reduces evictions (and therefore directory-induced invalidations) to negligible levels with arrays that are somewhat larger than the caches they are tracking [10].

ZCache implements the replacement process differently: it first retrieves all possible replacement candidates in a breadth-first fashion, selects the least desirable candidate using replacement policy information, and performs a few moves to evict that candidate and insert the new one. This process requires fewer lookups and far fewer moves than Cuckoo Directory (saving energy), can be pipelined (reducing latency), and enables using a replacement policy. For example, in a 4-way array, evaluating 52 replacement candidates requires 13 lookups and at most 2 moves in a zcache, but 17 lookups and 16 moves in a Cuckoo Directory. However, the Cuckoo Directory replacement process can stop early, while zcaches expand a fixed number of candidates. SCD combines the best features from both schemes to implement its replacement process.

Finally, due to the multiple hash functions and replacement process, these arrays can be characterized with simple, workload-independent analytical models that are accurate in practice. Prior work leverages these models to show that associativity depends only on the number of replacement candidates, not ways [25], and to implement scalable and efficient cache partitioning [26]. In this paper, we extend these models.
to characterize and show how to provision directories implemented with these arrays, including SCD and sparse directories, formalizing the empirical results of Ferdman et al. [10].

3. Scalable Coherence Directory

To scale gracefully, SCD exploits the insight that a directory does not need to provide enough capacity to track a specific number of addresses, but a specific number of sharers, ideally as many as can fit in the tracked caches. However, sparse directories use address-indexed arrays, so they use one directory tag per address. Instead, SCD represents sharer sets using a variable number of tags per address. Lines with one or a few sharers use a single directory tag with a limited pointer format, and widely shared lines employ a multi-tag format using hierarchical bit-vectors. We first describe the array used to hold the directory tags, then explain how SCD represents and operates on sharer sets.

3.1. SCD Array

Figure 1 shows the structure of the SCD array. It is similar to skew-associative caches, zcaches and Cuckoo Directories, with one hash function per way. However, hash functions take the concatenation of the line address and an index as input instead of using just the address. Every line in the directory will have a tag with index 0. Additionally, lines that use more than one directory tag will have those additional tags at locations with indexes other than 0. These indexes need not be consecutive, and are included in the hash functions so that multiple tags representing the same line map to different sets.

SCD’s replacement process is very similar to zcache’s, since it is more efficient than Cuckoo Directory’s, as explained in Section 2.2. However, SCD does not pipeline the replacement process, and stops looking for candidates as soon as an empty tag is found. As we will see, this greatly improves directory efficiency. SCD can optionally implement a replacement policy. However, replacement policies only make sense for underprovisioned directories — in Section 4 we will see that SCD can be sized to cause a negligible amount of evictions regardless of the workload, making a replacement policy unnecessary.

3.2. Line Formats

SCD encodes lines using different tag formats. Figure 2 illustrates these formats for a 1024-sharer directory. Lines with few sharers use a single-tag, limited pointer representation, with three pointers in the example. When more sharers are needed, SCD switches to a multi-tag format using hierarchical bit-vectors. In the example, a 32-bit root bit-vector tag indicates which subsets of cores share the line, while a set of 32-bit leaf bit-vectors encode the sharers in each subset. Leaf bit-vectors include a leaf number field that encodes the subset of sharers tracked by each leaf (e.g., leaf number 0 tracks sharers 0–31, 1 tracks 32–63, and so on). We will explain SCD’s operation using this two-level representation, but this can be easily extended to additional levels.

3.3. Directory Operations

SCD needs to support three operations on sharer sets: adding a sharer when it requests the line, removing a sharer when it writes back the line, and retrieving all sharers for an invalidation or downgrade.

Adding and removing sharers: On a directory miss, the replacement process allocates one tag for the incoming line with index 0 (possibly evicting another tag). This tag uses the limited pointer format. Further sharers will use additional pointers. When a sharer needs to be added and all the pointers are used, the line is switched to the multi-tag format as follows: First, the necessary bit-vector leaves are allocated.
for the existing pointers and the new sharer. Leaf tags are then populated with the existing and new sharers. Finally, the limited pointer tag transitions to the root bit-vector format, setting the appropriate bits to 1. Figure 3 illustrates this process. Removing a sharer (due to clean or dirty writebacks) follows the inverse procedure. When a line loses all its sharers, all its directory tags are marked as invalid.

**Invalidations and downgrades:** Invalidations are caused by both coherence (on a request for exclusive access, the directory needs to invalidate all other copies of the line) and evictions in the directory. Downgrades only happen due to coherence (a request for read on an exclusive line needs to invalidate all the sharers, if any). Coherence-induced invalidations are trivial: all the sharers are sent invalidation messages. If the address is represented in the hierarchical bit-vector format, all leaf bit-vectors are marked as invalid, and the root bit-vector tag transitions to the limited pointer format, which then encodes the index of the requesting core.

In contrast, eviction-induced invalidations happen to a specific tag, not an address. Limited pointer and root bit-vector tag evictions are treated like coherence-based invalidations, invalidating all sharers so that the tag can be reused. Leaf bit-vector evictions, however, only invalidate the subset of sharers represented in the tag. As we will see later, eviction-induced invalidations can be made arbitrarily rare.

**Additional levels and scalability:** SCD can use hierarchical bit-vector representations with more than two levels. A two-level approach scales to 256 sharers with ~16 bits devoted to track sharers (pointers/bit-vectors) per tag, 1024 sharers with ~32 bits, and 4096 sharers with ~64 bits. A three-level representation covers 4096 sharers with ~16 bits, 32768 sharers with ~32 bits, and 256K sharers with ~64 bits. Four-level implementations can reach into the millions of cores. In general, space and energy requirements increase with \( O(\log N) \), where \( N \) is the number of sharers, because the limited bit-vectors and the extended address space increase logarithmically. Since each tag needs on the order of 40-50 bits to store the line address anyway, having on the order of 16-32 bits of sharer information per tag is a reasonable overhead.

### 3.4. Implementation Details

SCD’s multi-tag format achieves the scalability of hierarchical directories, but since all tags are stored in the same array, it can be made transparent to the coherence protocol. We now discuss how to implement SCD to make it completely transparent to the protocol, and take the delay of additional accesses out of the critical path, providing the performance of a sparse directory.

**Scheduling:** Directories must implement some scheduling logic to make operations appear atomic to the protocol while ensuring forward progress and fairness. This is required in both sparse directories and SCD. For example, in a sparse directory adding a sharer is a read-modify-write operation, and the scheduling logic must prevent any intervening accesses to the tag between the read and the write (e.g., due to a conflicting request or an eviction). However, because SCD operations sometimes span multiple tags, ensuring atomicity is more involved. Note that the access scheduling logic makes the array type transparent to the directory: so long as the SCD array maintains atomicity, forward progress and fairness, it can be used as a drop-in replacement for a sparse array, with no changes to the coherence protocol or controller.

Our SCD implementation satisfies these goals with the following scheduling policies. First, as in conventional sparse directories, concurrent operations to the same address are serialized, and processed in FCFS order, to preserve atomicity and ensure fairness. Second, as in zcaches [25], the array is pipelined, and we allow concurrent non-conflicting lookups and writes, but only allow one replacement at a time. If the replacement process needs to move or evict a tag from an address of a concurrent request, it waits until that request has finished, to preserve atomicity. Third, similar to prior proposals using Cuckoo hashing where insertions are sometimes on the critical path [10, 19], we introduce an *insertion queue* to avoid the latency introduced by the replacement process. Tags are allocated in the insertion queue first, then inserted in the array. We have observed that, in practice, a 4-entry insertion queue suffices to hide replacement delay for sufficiently provisioned directories, where replacements are short, while severely underprovisioned directories require an 8-entry queue. Finally, to avoid deadlock, operations that require allocating new tags and block on a full insertion queue are not started until they allocate their space. This way, the replacement process is able to move or evict tags belonging to the address of the blocking request.

**Performance:** With this implementation, operations on a specific address are executed atomically once they start. Operations that require allocating one or more tags (adding a sharer) are considered completed once they have reserved enough space in the insertion queue. Writebacks, which require removing a sharer and never allocate, are considered complete once the root tag is accessed. Therefore, *adding and removing a sharer are typically as fast in SCD as in a conventional sparse directory*. On the other hand, coherence-induced invalidations on widely shared addresses need access several leaf tags to retrieve the sharers, invalidate them, and respond to the original requester once all invalidated sharers have responded. This could take longer with SCD than with a sparse directory, where the whole sharer set can be retrieved in one access (e.g., processing 1024 vs 32 sharers/cycle in our example). However, the critical-path latency of invalidations is determined by serialization latency in the network, as the directory can only inject one invalidation request per cycle, so SCD and sparse full-map directories perform similarly. Invalidation delays have a small performance impact in our simulations (Section 6), but should they become an issue, they can be reduced by processing invalidations in a hierarchical fashion, using multicast networks, or cruise-missile invalidates [2].

### 3.5. Storage Efficiency

We define *storage efficiency* as the average number of sharers that SCD encodes per tag. Storage efficiency determines how many directory tags are needed, and therefore how to size the directory. When all the lines have a single sharer, SCD has a storage efficiency of 1 sharer/tag. This is a common case (e.g., running a separate workload on each core, or a multithreaded workload where each working set is thread-
A simple optimization is to inspect entries among applications sharing the CMP. Therefore, we propose a limited pointer representation if possible. For example, using the format in Figure 2, a limited pointer tag with three sharers would have a storage efficiency of 3, while a fully shared line would have an efficiency of 1024 sharers/33 tags ≈ 31 sharers/tag. If the expected efficiency is consistently higher than 1, one could undersize or power off part of the directory and still achieve negligible invalidations. Note that SCD has a much lower dynamic range of storage efficiency than sparse directories (1-32 bits vs 1024 bits). For example, using this format in the example in Figure 2 would require 45 bits/tag for sharer information instead of 39 to hold the extra pointer (assuming we widen the leaf bit-vectors and narrow the root one).

### 4. Analytical Framework for Directories

We now show that directories built using zcache-like arrays in general, and SCD in particular, can be characterized with analytical models. First, we show that the fraction of replacements that result in evictions and the distribution of lookups per replacement is a function of the directory’s occupancy, i.e., the fraction of directory tags used. Second, although directory occupancy is time-varying and workload-dependent, we show that it can be easily bounded. Together, these results show that, with a small amount of overprovisioning, SCD can be designed to guarantee negligible invalidations and high energy efficiency in the worst case.

**Uniformity assumption:** In our analytical models, we rely on the assumption that the candidates visited in the replacement process have an uniform random distribution over the cache array. We have shown that in practice, this is an accurate assumption for zcaches [25, 26]. We leverage this assumption in the derivations, and verify its accuracy in Section 6 using simulation.

**Evictions as a function of occupancy:** Assume the directory has $T$ tags, of which $U$ are used. We define directory occupancy as $occ = U/T$. Per the uniformity assumption, replacement candidates are independent and uniformly distributed random variables, i.e., $cand_i \sim U[0, T - 1]$, and the probability of one being used is $Prob(cand_i used) = occ$. If the replacement process, as explained in Section 2.2, is limited to $R$ replacement candidates, the probability that all candidates are being used and we are forced to evict one of them is simply:

$$P_{ev}(occ) = \prod_{i=0}^{R-1} Prob(cand_i used)$$

Figure 4 plots this probability in linear and semi-logarithmic scales. Note that, with a reasonably large $R$, the eviction probability quickly becomes negligible. For example, with $R = \ldots$
64, \( P_{ev}(0.8) = 10^{-6} \), i.e., only one in a million replacements will cause an eviction when the directory is 80% full.

**Lookups per replacement:** We now derive the distribution and average number of lookups per replacement as a function of occupancy and the number of ways, \( W \). While Equation 1 characterizes worst-case behavior, this illustrates average behavior, and therefore average latency and energy requirements of the directory. First, the probability that all lines are occupied in a single lookup (\( W \) ways) is \( p = occ^W \). Second, the maximum number of lookups is \( L = R/W \). Therefore, the probability of finding an empty line in the \( k^{th} \) lookup is \( p_k = (1 - p)p^{k-1}, k \leq L \). Also, the probability of doing \( L \) lookups and not finding any empty line is \( P_{ev} \). Therefore, the average number of lookups is:

\[
\text{AvgLookups}(occ) = \sum_{k=1}^{L} k(1-p)p^{k-1} + L \cdot P_{ev}
\]

\[
= \frac{1 - pL}{1 - p} = \frac{1 - occ^R}{1 - occ^W} \quad (2)
\]

Figure 5 plots this value for different numbers of replacement candidates. Fortunately, even for high occupancies, the average number of lookups is much lower than the worst case (\( R/W \)). In fact, when evictions are negligible, the average is almost independent of \( R \), and is simply \( 1/(1 - p) = 1/(1 - occ^W) \). Therefore, assuming that we design for a negligible number of evictions, the maximum number of candidates \( R \) is irrelevant in the average case. In other words, a reasonable design methodology is to first define the target occupancy based on how much extra storage we want to devote versus how expensive allocation operations are, then set \( R \) high enough to satisfy a given eviction probability \( P_{ev} \).

**Bounding occupancy:** Occupancy is trivially bounded by 1.0 (the directory cannot use more lines than it has). However, if we can bound it to a smaller quantity, we can guarantee a worst-case eviction probability and average lookups independently of the workload. In general, the number of used tags is \( U = load/eff \), where \( load \) is the number of sharers that need to be tracked, and \( eff \) is the storage efficiency. Therefore, \( occ = U/T = \frac{load/T}{Sh} \). We can bound storage efficiency to \( eff \geq 1.0 \) sharers/line (Section 3.5). With a single-banked directory, the worst-case load is trivially the aggregate capacity of the tracked caches (in lines), which we denote \( C \). Therefore, if we never want to exceed a worst-case occupancy \( maxOcc \), we should size the directory with \( T = C/maxOcc \) tags. This in turn limits \( P_{ev} \) and \( AvgLookups \). For example, to ensure that the occupancy never exceeds 90%, we would need to overprovision the directory by 11%, i.e., have 11% more tags than lines are tracked, and with a 4-way, 64-replacement candidate array, this would yield worst-case \( P_{ev}(0.9) = 10^{-3} \) and worst-case \( AvgLookups(0.9) = 2.9 \) lookups/replacement. If we wanted a lower bound on \( P_{ev} \) (e.g., to provide stricter non-interference guarantees among competing applications sharing the CMP), we could use \( R = 128 \), which would give \( P_{ev} = 10^{-5} \), and still require 2.9 average lookups. Furthermore, most applications will not reach this worst-case scenario, and the directory will yield even better behavior. Alternatively, designers can provision the directory for an expected range of occupancies instead of for the worst case, reducing guarantees but saving storage space. In contrast, set-associative directories need to be overprovisioned by 2x or more to reduce evictions, and provide no guarantees [10].

When directories are banked, as it is commonly done with large-scale designs, this bound needs to be relaxed slightly, because the tracked caches will impose a different load on each bank. If a reasonable hash function is used, distributing addresses uniformly across the \( K \) directory banks, from basic probability theory, \( load \) has a binomial distribution \( \sim B(C, 1/K) \), with mean \( C/K \) and standard deviation \( \sqrt{C/K \cdot (1-1/K)} \). Therefore, the lower the number of banks, the more concentrated these values will be around the mean. In the CMP we study (\( C = 2^{31} \) lines, \( K = 64 \) banks), the standard deviation is only 0.5% of its mean, and it can be assumed that the worst-case load \( \approx C/K \) is constant across banks. In general, both \( P_{ev} \) and the number of lookups can be treated as functions of random variable \( C \) to determine the exact bounds for a given amount of overprovisioning.

In summary, we have seen that SCD can be characterized with analytical models, and can be tightly sized: high occupancies can be achieved with efficient replacements and incurring a negligible amount of evictions. These models apply to SCD and regular sparse directories implemented with arrays where the uniformity assumption holds (skew-associative caches, zcaches or Cuckoo Directories). We will show that these models are accurate in practice using simulation.

5. Experimental Methodology

**Modeled system:** We perform microarchitectural, execution-driven simulation using an x86-64 simulator based on Pin [23], and model a large-scale CMP with 1024 cores, shown in Figure 6. Table 2 summarizes its main characteristics. Each core is in-order and single-threaded, modeled after Atom [11], and has split 32KB L1 instruction and data caches and a private, inclusive, 128KB L2. All cores share a 256MB L3, which is kept coherent using a MESI coherence protocol. The CMP is divided in 64 tiles, each having 16 cores, a directory and L3 bank, and a memory controller. Both L2 and L3 are 4-way zcaches [25] with 16 and 52 replacement candidates, respectively. Caches and directories use \( H_3 \) hash functions, which are simple to implement and work well in practice [5, 25]. Tiles are connected with an 8×8 mesh network-on-chip (NoC) with physical express links.

The system we model is in line with several large-scale CMP proposals, such as Rigel [17, 18] and ATAC [20], and represents a reasonable scale-up of commercial designs like Tilera’s Gx-3100 [30], which has 100 cores and 32 MB of distributed, directory-coherent last-level cache that can be globally shared, and is implemented at 40 nm. We estimate that our target CMP should be implementable at 14 or 11 nm. Using McPAT [22], we find that a scaled-down version of this system with 8 tiles and 128 cores would require 420 mm² and 115 W at 32 nm. We use the component latencies of this scaled-down CMP in the 1024-core simulations.

**Directory implementations:** We compare three different directory organizations: sparse, sparse hierarchical (two-level), and SCD. The classic sparse organization has a full-map 1024-bit sharer vector per line. The hierarchical implementation has a distributed first directory level every two tiles,
and a second, banked directory level. Therefore, both levels have 32-bit sharer vectors. SCD has the same organization as shown in Figure 2, with 3 limited pointers and a 32/32 2-level bit-vector organization. All organizations nominally use 4-way zcache arrays with 52 replacement candidates and $H_3$ hash functions, so the sparse organization is similar to a Cuckoo Directory [10]. All directories are modeled with a 5-cycle access latency. We compare directories with different degrees of coverage. Following familiar terminology for TLBs, we define coverage as the maximum number of addresses that can be represented in the directory, as a percentage of the total lines in the tracked caches. Therefore, 100%-coverage Sparse and SCD have as many tags as lines in the tracked caches, while a hierarchical directory with 100% coverage has twice as many tags (as each address requires at least two tags, one per level).

Workloads: We simulate 14 multithreaded workloads selected from multiple suites: PARSEC [3] (blackscholes, canneal, fluidanimate), SPLASH-2 [32] (barnes, fft, lu, ocean, radix, water), SPECCOMP (applu, equake, wupwise), SPECJBB2005 (specjbb), and BioParallel [16] (svm). We have selected workloads that scale reasonably well to 1024 cores and exhibit varied behaviors in the memory hierarchy (L1, L2 and L3 misses, amount of shared data, distribution of sharers per line, etc.). We simulate complete parallel phases (sequential portions of the workload are fast-forwarded), and report relative execution times as the measure of performance. Runs have at least 200 million cycles and 100 billion instructions, ensuring that all caches are warmed up. We perform enough runs to guarantee stable averages (all results presented have 95% confidence intervals smaller than 1%).

6. Evaluation

6.1. Comparison of Directory Schemes

Directory size and area: Table 3 shows the directory size needed by the different directory organizations (SCD, Sparse, and Hierarchical) for 128 to 1024 cores. We assume line addresses to be 42 bits. Storage is given as a percentage of total tracked cache space. All directories have 100% coverage.

As we can see, SCD significantly reduces directory size. A 2-level SCD uses $3 \times 13 \times$ less space than a conventional sparse directory, and around $2 \times$ less than a 2-level hierarchical implementation. A 3-level SCD would be even more efficient (e.g., requiring 18 bits of coherence data per tag instead of 39 at 1024 cores), although gains would be small since the address field would take most of the tag bits.

We can approximate directory area using directory size and assuming the same storage density for the L3 cache and the directory. On our 1024-core CMP, SCD would require $20.2$ MB of total storage, taking $3.1%$ of die area, while a two-level hierarchical directory would require $39.5$ MB, taking $6.1%$ of die area. Sparse directories are basically unimplementable at this scale, requiring $267$ MB of storage, as much as the L3 cache.

Performance: Figure 7 compares execution time, global NoC traffic and average memory access time among different directory organizations. Each directory is simulated at both 100% and 50% coverage. Smaller values are better for all graphs, and results are normalized to those of an idealized directory (i.e., one with no invalidations). Recall that all directory organizations use 4-way/52-candidate zcache arrays.

We will discuss set-associative arrays in Section 6.4.

Looking at Figure 7a, we see that both SCD and Sparse achieve the performance of the ideal directory in all applications when sized for 100% coverage, while their 50%-sized variants degrade performance to varying degrees (except on canneal, which we will discuss later, where performance increases). Underprovisioned Sparse directories perform slightly better than SCD because their occupancy is lower, as they require one line per address. Hierarchical directories, on the other hand, are slightly slower even at 100%.
coverage, as they require an additional level of lookups, and their performance degrades significantly more in the undersized variant. Note that the 50%-coverage Hierarchical directory has about the same area as the 100%-coverage SCD.

Figures 7b and 7c give more insight into these results. Figure 7b breaks down NoC traffic into GET (exclusive and shared requests for data), PUT (clean and dirty writebacks), coherence INV (invalidation and downgrade traffic needed to maintain coherence), and eviction INV (invalidations due to evictions in the directory). Traffic is measured in flits. We see that all the 100%-sized directories introduce practically no invalidations due to evictions, except SCD on canneal, as canneal pushes SCD occupancy close to 1.0 (this could be solved by overprovisioning slightly, as explained in Section 4). The undersized variants introduce significant invalidations. This often reduces PUT and coherence INV traffic (lines are evicted by the directory before the L2s evict them themselves or other cores request them). However, those evictions cause additional misses, increasing GET traffic. Undersized directories increase traffic by up to 2×.

Figure 7c shows the effect that additional invalidations have on average memory access time (AMAT). It shows normalized AMAT for the different directories, broken into time spent in the L2, local directory (for the hierarchical organization), NoC, directory and L3, coherence invalidations, and main memory. Note that the breakdown only shows critical-path delays, e.g., the time spent on invalidations is not the time spent on every invalidation, but the critical-path time that the directory spends on coherence invalidations and downgrades. In general, we see that the network and directory/L3 delays increase, and time spent in invalidations decreases sometimes (e.g., in fluidanimate and canneal). This happens because eviction invalidations (which are not on the critical path) reduce coherence invalidations (on the critical path). This is why canneal performs better with underprovisioned directories: they invalidate lines that are not reused by the current core, but will be read by others (i.e., canneal would perform better with smaller private caches). Dynamic self-invalidation [21] could be used to have L2s invalidate copies early and avoid this issue.

In general, we see that hierarchical directories perform much worse when undersized. This happens because both the level-1 directories and level-2 (global) directory cause invalidations. Evictions in the global directory are especially troublesome, since all the local directories with sharers must be invalidated as well. In contrast, an undersized SCD can prioritize leaf or limited pointer lines over root lines for eviction, avoiding expensive root line evictions.

Energy efficiency: Due to a lack of energy models at 11 nm, we use the number of array operations as a proxy for energy efficiency. Figure 8 shows the number of operations (lookups...
and writes) done in SCD and Sparse directories. Each bar is normalized to Sparse. Sparse always performs fewer operations because sharer sets are encoded in a single line. However, SCD performs a number of operations comparable to Sparse in 9 of the 14 applications. In these applications, most of the frequently-accessed lines are represented with limited pointer lines. The only applications with significant differences are barnes (5%), svm, fluidanimate (20%), lu (40%) and canneal (97%). These extra operations are due to two factors: first, operations on multi-line addresses are common, and second, SCD has a higher occupancy than Sparse, resulting in more lookups and moves per replacement. However, SCD lines are narrower, so SCD should be more energy-efficient even in these applications.

6.2. SCD Occupancy

Figure 9 shows average and maximum used lines in an ideal SCD (with no evictions), for different SCD configurations: 1 to 4 limited pointers, with and without coalescing. Each bar shows average occupancy, and is broken down into the line formats used (limited pointer, root bit-vector and leaf bit-vector). Results are given as a fraction of tracked cache lines, so, for example, an average of 60% would mean that a 100%-coverage SCD would have a 60% average occupancy assuming negligible evictions. These results show the space required by different applications to have negligible evictions.

In general, we observe that with one pointer per tag, some applications have a significant amount of root tags (which do not encode any sharer), so both average and worst-case occupancy sometimes exceed 1.0×. Worst-case occupancy can go up to 1.4×. However, as we increase the number of pointers, limited pointer tags cover more lines, and root tags decrease quickly (as they are only used for widely shared lines). Average and worst-case occupancy never exceed 1.0× with two or more pointers, showing that SCD’s storage efficiency is satisfactory. Coalescing improves average and worst-case occupancy by up to 6%, improving workloads where the set of shared lines changes over time (e.g., water, svm, canneal), but not benchmarks where the set of shared lines is fairly constant (e.g., fluidanimate, lu).

6.3. Validation of Analytical Models

Figure 10 shows the measured fraction of evictions (empirical $P_{ev}$) as a function of occupancy, on a semi-logarithmic scale, for different workloads. Since most applications exercise a relatively narrow band of occupancies for a specific directory size, to capture a wide range of occupancies, we sweep coverage from 50% to 200%, and plot the average for a specific occupancy over multiple coverages. The dotted line shows the value predicted by the analytical model (Equation 1). We use 4-way arrays with 16, 52 and 104 candidates. As we can see, the theoretical predictions are accurate in practice.

Figure 11 also shows the average number of lookups for the 52-candidate array, sized at both 50% and 100% coverage. Each bar shows the measured lookups, and the red dot shows the value predicted by the analytical model. Again, empirical results match the analytical model. We observe that with a 100% coverage, the number of average lookups is significantly smaller than the maximum ($R/W = 13$ in this case), as occupancy is often in the 70%-95% range. In contrast, the underprovisioned directory is often full or close to full, and the average number of lookups is close to the maximum.

In conclusion, we see that SCD’s analytical models are accurate in practice. This lets architects size the directory using simple formulas, and enables providing strict guarantees on directory-induced invalidations and energy efficiency with a small amount of overprovisioning, as explained in Section 4.
can be observed with sparse and hierarchical directories as well. In conclusion, if designers want to ensure negligible directory-induced invalidations and guarantee performance isolation regardless of the workload, directories should not be built with set-associative arrays. Note that using zcache arrays has more benefits in directories than in caches. In caches, zcache have the latency and energy efficiency of a low-way cache on hits, but replacements incur similar energy costs as a set-associative cache of similar associativity [25]. In directories, the cost of a replacement is also much smaller since replacements are stopped early.

7. Conclusions

We have presented SCD, a single-level, scalable coherence directory design that is area-efficient, energy-efficient, requires no modifications to existing coherence protocols, represents sharer sets exactly, and incurs a negligible number of invalidations. SCD exploits the insight that directories need to track a fixed number of sharers, not addresses,
by representing sharer sets with a variable number of tags: lines with one or few sharers use a single tag, while widely shared lines use additional tags. SCD uses efficient highly-associative caches that allow it to be characterized with simple analytical models, and enables tight sizing and strict probabilistic bounds on evictions and energy consumption. SCD requires $13 \times$ less storage than conventional sparse full-map directories at 1024 cores, and is $2 \times$ smaller than hierarchical directories while using a simpler coherence protocol. Using simulations of a 1024-core CMP, we have shown that SCD achieves the predicted benefits, and its analytical models on evictions and energy efficiency are accurate in practice.

Acknowledgements

We sincerely thank Christina Delimitrou, Jacob Leverich, David Lo, and the anonymous reviewers for their useful feedback on earlier versions of this manuscript. This work was supported in part by the Stanford Pervasive Parallelism Laboratory. Daniel Sanchez was supported by a Hewlett-Packard Stanford School of Engineering Fellowship.

References


Virtues and Limitations of Commodity Hardware Transactional Memory

Nuno Diegues, Paolo Romano, Luís Rodrigues
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
{nmld, paolo.romano, ler}@tecnico.ulisboa.pt

ABSTRACT

Over the last years Transactional Memory (TM) gained growing popularity as a simpler, attractive alternative to classic lock-based synchronization schemes. Recently, the TM landscape has been profoundly changed by the integration of Hardware TM (HTM) in Intel commodity processors, raising a number of questions on the future of TM.

We seek answers to these questions by conducting the largest study on TM to date, comparing different locking techniques, hardware and software TMs, as well as different combinations of these mechanisms, from the dual perspective of performance and power consumption.

Our study sheds a mix of light and shadows on currently available commodity HTM: on one hand, we identify workloads in which HTM clearly outperforms any alternative synchronization mechanism; on the other hand, we show that current HTM implementations suffer of restrictions that narrow the scope in which these can be more effective than state of the art software solutions. Thanks to the results of our study, we identify a number of compelling research problems in the areas of TM design, compilers and self-tuning.

Categories and Subject Descriptors
D.1.3 [Software]: Programming Techniques - Concurrent Programming

Keywords
Empirical Study; Synchronization Techniques; Transactional Memory; Performance; Energy Efficiency

1. INTRODUCTION

The advent of multi-core architectures has brought concurrent programming to the forefront of software development. For many years, locking has represented the de-facto standard approach to synchronization in concurrent applications. However, the inherent complexity and error-proneness of fine-grained locking [32] has motivated intense research on alternative methodologies aimed at making parallel programming accessible to the mass of software developers.

Transactional Memory (TM) [27] is one of the most prominent proposals in this sense. With the TM abstraction, programmers are required only to identify which code blocks should run atomically, and not how concurrent accesses to shared state should be synchronized (as with locks). The TM is then responsible for guaranteeing correctness by aborting transactions that would generate unsafe histories.

Over the last decade a large body of TM research focused on software-based implementations (STM) (e.g. [13, 17]). Unlike hardware implementations, however, STM requires software instrumentation of read and write memory accesses to trace conflicts between concurrent transactions. This instrumentation can, in certain scenarios, introduce large overheads and hinder performance with respect to conventional fine-grained locking [5]. HTM support is thus desirable, but its absence from commodity processors caused most research to be evaluated solely on simulators (e.g. [25]) — the only notorious exception being the Rock processor [12], which was never commercialized.

Recently, the maturing of TM research led to a breakthrough that changed drastically this scenario: two major market players, IBM and Intel, introduced HTM support in their latest processors [37, 3, 36], targeting, respectively, HPC and commodity systems. This represents a significant milestone for TM, mainly due to the predictable widespread availability of Intel Haswell processors, which bring HTM support to millions of systems ranging from high-end servers to common laptops.

The advent of HTM in commodity processors raises a number of questions concerning the future of TM and concurrent programming: how competitive are available HTMs when compared with state of the art STMs? Will the performance of HTM be sufficiently alluring to turn TM into a mainstream programming paradigm? What role will STM play now that HTM is so widely available? How limiting are the architectural restrictions of existing HTM designs?

In this paper we seek an answer to these questions by conducting the largest study on TM-based synchronization to date. We compare, from the twofold perspective of performance and energy-efficiency, a range of synchronization mechanisms: 6 lock based approaches with different granularities; 4 state of the art STMs; Intel TSX's implementation of HTM; and 2 Hybrid TMs (HyTM) that use STM and HTM mechanisms in synergy. We study highly heterogeneous applications, encompassing 1) STAMP, a de-facto standard suite of benchmarks for TM, 2) Memcached [35], a popular in-memory caching system that was recently ported.
to use TM, and 3) concurrent data structures that are widely used as building blocks of parallel applications (yet, hard to parallelize efficiently). The results of our study allow us to draw two main conclusions:

**Lights and shadows for HTM:** Approaches based on TSX yielded outstanding performance in workloads characterized by small transactions, such as concurrent data structures and Memcached, but only with two of the STAMP benchmarks. TSX performance is strongly dependent on the access patterns to L1 cache, and long running transactions can lead to frequent cache capacity exceptions and spurious aborts. When transaction-intensity is medium, TSX is only the best choice for a limited degree of parallelism, and it is generally better on the energy side than on the performance side. The impact of its hardware limitations are highlighted by several STAMP benchmarks that generate long transactions, and in which TSX is outperformed by both locking and STM solutions. On the other hand, TSX shines as a synchronization primitive for concurrent data structures, for which it is by far the best choice in all considered workloads, with speed-ups up to 3.3× over the best alternative scheme.

**STM is still competitive:** Our study also shows that STM is quite competitive as an all-around solution across benchmarks, workloads, and parallelism degrees. Although STM was initially proposed as a prototyping alternative to actual hardware implementations of TM, its evolution throughout a decade of intense research has resulted in several highly-optimized mechanisms, which achieve performance comparable to that of fine-grained locking. This does not mean that STMs embody a perfect solution; instead, this result highlights the current limitations of HTM support, which make of STM still the most robust solution to date.

Further, the results of our study unveil a number of critical issues related with HTM performance and allow for identifying several research problems, whose timely solution could significantly enhance the chances for HTM to turn into a mainstream paradigm for concurrent programming:

**HyTMs: a missed opportunity?** The outcome of our study for what concerns the efficiency of HyTMs, when employed in conjunction with Intel’s TSX, is rather grim. The mechanisms currently adopted to support the simultaneous coexistence of HTM and STM induce high overheads in terms of additional spurious aborts. Our study highlights that these costs make HyTM generally less efficient than solutions based purely on STM or TSX+locking. This motivates further research in the design of architectural supports (e.g., non-transactional memory accesses from transactions) capable of exploiting the potential synergies of HyTMs.

**Complexity of HTM tuning.** HTM performance can be significantly affected by the settings of several parameters and mechanisms. Without proper tuning, TSX suffer average throughput losses of 72% and of 89% in power consumption. Also, the optimal configuration of these parameters can vary significantly, depending on the characteristics of the workload. These findings urge for novel approaches capable of removing from the shoulders of programmers the burden of manually tuning HTM, by delegating this task to run-time or compiler based solutions.

**Relevance of selective instrumentation.** Both TSX and GCC library for STM trace every memory access performed within a transaction. We show that this can cause significant increases of the transaction footprint’s size, amplifying the instrumentation overheads in STM, and the chances of incurring in capacity exceptions in HTM. These results motivate research on cross-layer mechanisms operating at the compiler and at the hardware level, aimed to achieve selective instrumentation in a way that is both convenient for the programmer and efficiently implementable in hardware.

The rest of the paper is structured as follows. Section 2 discusses related work. Section 3 overviews the synchronization mechanisms considered in the study. Section 4 describes our methodology, after which Section 5 presents preliminary experiments aimed at tuning properly TSX. We then present our study in Sections 6-7. In Section 8 we identify several research questions suggested by the findings of our study. Finally, Section 9 concludes the paper.

### 2. RELATED WORK

Transactional Memory was initially proposed as an extension to multi-processors’ cache coherence protocols [27]. Due to the difficulty of rapid prototyping in hardware environments, researchers resorted to STMs to advance the state of the art [18, 17, 16, 14]. Simultaneously, hardware-based implementations have also been proposed, whose designs were validated using simulators [25].

The concern for both performance and power consumption metrics has been only marginally explored in the scope of TM, and mostly relying on simulation studies that did not target Intel’s architecture (whose internals are only partially disclosed). In both [21, 20] the authors assess the behaviour of different HTM implementations via simulation (the latter focusing on embedded systems). The approach was also taken by [2], where the power consumption of one STM was studied via simulation. More recently, [22] studied both power consumption and performance in a non-simulated environment. Yet, this work considered a restricted set of synchronization alternatives focusing mainly on one STM.

As already mentioned, both Intel [37] and IBM [3, 36] have integrated HTM in their processors. IBM processors target high performance computing infra-structures that are not expected to be used in commodity systems. In this work, we focus on the former (by Intel), for which our results show aspects and insights that were not highlighted by Intel’s paper [37]. Before the recent release of Intel Haswell processors, researchers had already proposed some theoretical improvements to best-effort HTMs [1, 30]. We integrate these mechanisms in our HTM-based systems, and evaluate them for the first time using an actual HTM implementation.

Traditional lock-based synchronization techniques have been thoroughly studied throughout decades. In [19], the authors show that the power consumption of locking primitives can be improved by exploring a trade-off between processor deep sleeping states, frequency downsizing and busy waiting. We highlight a recent work [16], which studied the impact in performance of different lock designs and hardware architectures (without however considering TM).

Our work is also related to the body of literature on performance modeling of TMs, which have relied on methodologies such as analytical modeling [11, 26], as well as machine learning [34]. These proposals were applied to self-tune various TM parameters, and our results clearly indicate the potentiality and importance of this line of research also in the scope of HTM.
3. SYNCHRONIZATION MECHANISMS CONSIDERED IN THE STUDY

In this comparative study we considered the several synchronization mechanisms listed in Table 1:

**Locks** — Decades of research on lock-based synchronization have resulted in a plethora of different implementations, many times trading off subtle changes with great impact in performance. We consider 6 different lock implementations [10] and both coarse and fine-grained strategies. Contrarily to the other approaches we used, fine-grained locking requires a per-application lock allocation strategy, which is a non-generalizable and error-prone task [32].

**STM** — With STM, reads and writes to shared memory (inside atomic blocks) are instrumented to detect conflicts between transactions. This instrumentation induces overheads that can have a detrimental impact on the efficiency of STMs. Yet, much research has been devoted over the last years to reduce STM’s overheads. For our study we selected four state of the art STMs, which are representative of different choices in the design space of TM. These include an STM optimized for validations at commit-time (TL2 [13]); to maximize performance at low thread counts (NOrec [7]); in high contention scenarios (SwissTM [17]); and to minimize instrumentation costs (TinySTM [18]).

**HTM** — HTM implements a concurrency control scheme in hardware, avoiding the overheads of STM instrumentation. In our study we consider Intel TSX’s implementation of HTM, which is integrated in the family of Intel commodity processors (Haswell). One fundamental design principle of TSX (and in general of HTM) is its best-effort nature: one cannot depend exclusively on TSX to synchronize accesses to data, since a transaction is not guaranteed to commit, even in absence of contention. Briefly, TSX uses the L1 cache to buffer transactional writes, and relies on the cache coherence protocol to detect conflicts. A plausible reason for a transaction to fail in hardware is because its data footprint exceeds the L1 capacity. Hardware transactions are also subject to abort due to reasons like page faults and system calls.

As a result, a fallback software synchronization mechanism must be provided to ensure progress in case a transaction cannot be committed with HTM. The software fallback mechanism must co-operate with the hardware in order to ensure correctness. As we shall see, the mechanisms used to coordinate the execution of the fallback mechanism have a crucial impact on the performance of TSX. The simplest approach, as suggested by Intel’s optimization manual, is to use a single lock to protect atomic blocks (we call it TSX-GL).

When a hardware transaction aborts, it has the alternative to acquire the global lock instead. To ensure a correct interplay with the fallback, hardware transactions must read the lock as free to guarantee correctness; the transactional semantics will guarantee that the transaction commits only when there is no ongoing fallback execution.

An obvious extension of this idea is to use fine-grained locks (TSX-FL). As the TM abstraction is motivated by the need of relieving programmers from the complexity of designing locking schemes, the usage of fine-grained locks as a fallback for HTM sounds somewhat contradictory. However, this choice allows us to assess to what extent a simplistic fallback (using a single lock) can hinder parallelism. Also, fine-grained locks may be automatically crafted, to some extent, by using recent techniques based on static analysis [29].

**HyTM** — Another mechanism proposed in the literature is to use an STM as fallback for HTM, also known as Hybrid TM (HyTM). Its main advantage is to allow concurrent execution of hardware transactions and software ones, used in the fallback. However, during their concurrent execution, both software and hardware transactions have to play along in order to preserve correctness. In our study we considered two state of the art Hybrid TM proposals [30, 8], which are evaluated for the first time on a commodity HTM. We exploit the idea of reduced hardware transactions: normally transactions execute mainly in hardware, with a pure software fallback; the idea of these HyTMs is to have an intermediary mode where the software fallback still relies partially on hardware speculations to boost performance, namely during the commit in the fallback.

4. METHODOLOGY AND TESTBED

We consider in our study several parallel applications using atomic blocks for synchronization. First we use the STAMP suite, a popular set of benchmarks for TM [4], encompassing 8 applications representative of various domains that generate highly heterogeneous workload domains. We excluded the Bayes application given its non-deterministic executions, and used standard parameters for each application. STAMP contains manually instrumented reads and writes inside atomic blocks to invoke the STM-based synchronization. Naturally, this is not relevant for the case of pure HTM approaches. We shall additionally present results for compiler based instrumentation in Section 8.

We also consider a red-black tree and a hashmap, as examples of concurrent data structures, which represent important building blocks of parallel applications and that have two interesting characteristics: they are very hard to parallelize efficiently using locking schemes, and are challenging for STMs given that they generate extremely short transactions that suffer from relatively large instrumentation overheads. Finally, we also used a recent TM-based porting of the popular Memcached [35], an in-memory cache, widely used for instance at Facebook [31].

Each experiment is the average of 20 executions. We use the geometric mean whenever we show an average of normalized results. We often show speedup results, which are relative to the performance of sequential, non-instrumented executions, unless stated otherwise. The reported measurements of power consumption were obtained via the Intel RAPL [9] facility and are restricted to the processor and memory subsystems. Recent studies [23, 24] show that the model used by Intel RAPL estimates quite accurately the power consumption, when compared to a power meter attached to the machine.

Our machine is equipped with an Intel Haswell Xeon E3-1275 3.5GHz processor and 32GB RAM. This choice is dic-

<table>
<thead>
<tr>
<th>Table 1: Mechanisms used for synchronization in our study.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mechanism</strong></td>
</tr>
<tr>
<td><strong>Locks</strong></td>
</tr>
<tr>
<td><strong>STMs</strong></td>
</tr>
<tr>
<td><strong>HTMs</strong></td>
</tr>
<tr>
<td><strong>HyTMs</strong></td>
</tr>
</tbody>
</table>
tated by the requirement of using a processor equipped with TSX, which is limited for now to 4 cores (and 8 hyper-threads). We always pin threads to physical cores in a round-robin fashion; for instance, 4 threads will be allocated uniformly, one per core. As a result, hyper-threading is only used when 5 or more threads are used. We used GCC 4.8.1 with all compiler optimizations enabled and Ubuntu 12.04.

5. TUNING TSX FALLBACK PATH

Before comparing the considered synchronization mechanisms, we conduct a set of preliminary experiments evaluating several alternative configurations of the coupling between TSX and its fallback. As we shall see, this can have significant impact on TSX’s efficiency. The settings identified thanks to this preliminary study will be adopted in the remainder of the paper to ensure that the comparison is performed using an appropriately tuned HTM.

We begin in Section 5.1 by comparing the performance and energy efficiency when using six locks implementations to implement the fallback mechanism of TSX. This shall allow us to narrow down the multitude of combinations of TSX and lock implementations assessed in our study. Next, in Section 5.2, we optimize TSX-GL with a recently proposed technique [1] aimed at reducing spurious hardware aborts. This shall provide some insights about its actual practical effectiveness, as it was never evaluated before. Lastly, we investigate when it is best to give up on hardware and trigger the fallback path, in Section 5.3.

5.1 The Impact of Locks on the Fallback

The simpler way to use TSX is by relying on very coarse-grained locks on the fallback, or simply a single one. We considered the six lock listed in Table 1, to be used in the fallback. These implementations are representative of different design choices, and our goal is to understand if there is some implementation that consistently performs above the average across all parallelism degrees and benchmarks.

Table 2 shows the performance of TSX given the backing lock implementation used in the fallback path. We show the average overhead with respect to the best performing lock in each experiment, considering both time to complete the benchmark as well as power consumed. The reported overhead is the average across all STAMP benchmarks and thread counts (1 to 8). Using this metric, we can see that the Ticket, MCS and CLH locks perform best.

For each benchmark and thread count, we additionally sorted the considered lock implementations according to either their performance or power consumption, determining in this way their rank for that benchmark/configuration. This shows that no lock implementation is always the best or worse. However, we can see that the Ticket lock is consistently ranked higher, for which reason we shall rely on it from now on whenever we require locking (both standalone, or in the fallback of TSX).

5.2 Improving the Single-lock Fallback

The recommended fallback mechanism for TSX relies on a global lock (as explained in Section 3). In this section we evaluate for the first time on TSX the efficiency of an alternative technique proposed to reduce the situations under which best-effort HTMs have to follow this pessimistic fallback path [1].

The idea is that it may not be safe for transactions to execute speculatively in hardware if, at the same time, some transaction is executing pessimistically after acquiring the global lock. A pessimistic execution cannot restart, and hence its accesses must be consistent when faced with uncertainty. For this reason, TSX’s usage guide points out that the lock has to be read as being free during the course of a hardware transaction. Then any transaction that activates the fallback path has to first acquire the lock, and cause every hardware transaction to abort. This can cause a chain effect, also known as lemming effect [12], where the aborted hardware transactions also try to acquire the lock, preventing hardware speculation from ever resuming.

In [1], the authors use an auxiliary lock to prevent the lemming effect. The idea is to guard the global lock acquisition by another lock. Aborted hardware transactions have to acquire this auxiliary lock before restarting speculation, which effectively serializes them. However, this auxiliary lock is not added to the read-set of hardware transactions, which avoids aborting concurrent hardware transactions. If this procedure is attempted sometimes before actually giving up and acquiring the global lock, then the chain reaction effect can be avoided: the auxiliary lock serves as a manager preventing hardware aborts from continuously acquiring the fallback lock and preventing hardware speculations.

In Table 3 we compare TSX using the auxiliary lock against a single-lock (TSX-GL). For this, we report values for time, energy, and Energy Delay Product (EDP), normalized with respect to TSX-GL (analogously to traditional speedup metrics). We report the average across either benchmarks or threads. Naturally, we can see that there is no difference with 1 thread because there is no concurrency and hence no problem resuming speculative execution. But beyond that, and in particular at larger concurrency levels, this technique...

<table>
<thead>
<tr>
<th>Lock</th>
<th>Overhead (%)</th>
<th>Rank</th>
<th>Overhead (%)</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ticket</td>
<td>1.0</td>
<td>1.75</td>
<td>1.1</td>
<td>1.75</td>
</tr>
<tr>
<td>MCS</td>
<td>2.4</td>
<td>2.62</td>
<td>1.2</td>
<td>2.25</td>
</tr>
<tr>
<td>CLH</td>
<td>2.9</td>
<td>3.62</td>
<td>2.4</td>
<td>3.38</td>
</tr>
<tr>
<td>PW</td>
<td>14.2</td>
<td>4.89</td>
<td>17.4</td>
<td>3.88</td>
</tr>
<tr>
<td>TTAS</td>
<td>15.2</td>
<td>5.00</td>
<td>17.4</td>
<td>4.88</td>
</tr>
<tr>
<td>Spin</td>
<td>16.4</td>
<td>5.00</td>
<td>17.5</td>
<td>4.88</td>
</tr>
</tbody>
</table>

Table 3: Normalized performance of auxiliary lock [1] over TSX-GL across benchmarks and threads (higher is better).
helps consistently to improve the EDP. Some benchmarks do not show any difference because there are very little aborts (SSCA2) or TSX is not able to execute speculatively most of the time (Labyrinth). Yada’s workload is conflict-intensive, for which reason the non-optimized approach is slightly better due to its inherent pessimism in following the fallback path — that pays off since the high conflict probability limits the effectiveness of optimistic transactions.

5.3 Retry Policy for the Fallback

Given that TSX must always have a fallback due to its best-effort nature, an important decision is when to trigger that path. Upon a transaction abort, TSX provides an error code that informs about the reason of the abort. An abort due to a capacity exception is typically a good reason to trigger the fallback path. However, hardware transactions may abort for various micro-architectural conditions that are less deterministically prone to happen upon transaction re-execution, and even capacity exceptions may not always be deterministic. Also, of course, transactions may abort due to data contentions. In these situations one may aggressively trigger the fallback, or opt to insist on using HTM.

As we will shall discuss in more detail in Section 8, the optimal choice of the retry policy can vary significantly across workloads and degrees of parallelism. As it is impractical to assume that the retry policy is ad-hoc tuned by programmers for each and single workload/application, we set the number of retries to 5, which is the configuration reported to deliver best all-around performance with TSX [37, 28] (a result that we have confirmed with TSX-GL on our testbed). For the HyTMs, 4 times was found to be the best number of retries on average.

6. STAMP BENCHMARK SUITE

In this section we rely on the STAMP benchmark suite to assess the efficiency of all the synchronization mechanisms listed in Section 3, namely HTM, STMs, HyTMs, and locking. In the following, we shall always include the TSX optimizations discussed in the previous section.

We start by summarizing our results in Table 4. There, we list the STAMP benchmarks sorted by two important characteristics of their workloads: the contention level between transactions, and the percentage of the workload that is transactional. We then identify the mechanism that takes the least time to complete and which one consumes the least power, given the averaged results across threads.

This summarized perspective allows to highlight an interesting fact. It is possible to distinguish three categories in which TSX behaves differently, according to the transaction’s characteristics. Kmeans and SSCA2 represent workloads with small transactions, medium frequency and low contention; here, TSX-GL performs consistently better than the alternatives across all threads. Intruder and Vacation exhibit medium profiles for what concerns the time spent in transactions and contention; in these cases, TSX-GL results in the best performing solution using up to 4, resp. 2, threads, and the most energy efficient up to 5, resp. 4, threads. Finally, the other benchmarks spend almost all the time in transactions, encompassing both low and high contentions scenarios. In these settings, TinySTM emerges as the most robust solution, both from the perspective of energy and performance.

This analysis allows to draw a set of guidelines to select which synchronization to use, at least when considering applications having analogous characteristics to those included in the STAMP suite. TSX-GL is desirable when transactions are small, generate low/medium contention, and the application does not spend all the time executing transactions. When contention increases, or the frequency of transactions is high, TSX-GL is competitive up to a medium degree of parallelism. In the remaining cases, STM is often the best choice, even when compared with fine-grained locking. The considered HyTMs perform poorly compared to the alternatives, never clearly outperforming the competing schemes in any benchmark. In Sections 6.1-6.2 we present our experiments with STAMP. We will consider additional benchmarks and fine-grained locks in Section 7.

6.1 Performance Study

In Fig. 1 we show, for each benchmark and while varying the parallelism level, the speedup of all the considered synchronization schemes (with the exception of schemes based on fine-grained locking, which shall be presented in Section 7) with respect to a sequential, non-instrumented execution, and the power consumption during the execution (in Kilo Joules). This allows us to discuss in detail the differences between the mechanisms in different workloads.

Kmeans: This benchmark yields the biggest gap in performance between a TSX variant and STMs. Namely, TSX-GL reaches 3.5× speedup over a sequential execution, beating every other alternative both performance-wise and in terms of energy-efficiency. An interesting trend concerning energy-efficiency is that the power consumption with TSX (and, to some extent, also for all other synchronization schemes but GL) tends to slightly decrease as the parallelism level grows, which is a symptom of efficient utilization of the available architecture resources achievable using TM-based solutions. If we consider TSX-TL2 and TSX-NOrec, they are still com-

Table 4: Summary of results according to the workload characterization of the STAMP suite ($xt$ = number of threads).

<table>
<thead>
<tr>
<th></th>
<th>Time in Tx (%)</th>
<th>Contention</th>
<th>Best Performing</th>
<th>Least Power Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>kmeans</td>
<td>low (7)</td>
<td>low</td>
<td>TSX-GL</td>
<td>TSX-GL</td>
</tr>
<tr>
<td>scca2</td>
<td>low (17)</td>
<td>low</td>
<td>TSX-GL</td>
<td>TSX-GL</td>
</tr>
<tr>
<td>intruder</td>
<td>medium (33)</td>
<td>high</td>
<td>TSX-GL ≤ 4t; TinySTM ≥ 5t</td>
<td>TSX-GL ≤ 5t; TinySTM ≥ 6t</td>
</tr>
<tr>
<td>vacation</td>
<td>high (89)</td>
<td>low</td>
<td>TSX-GL ≤ 3t; TinySTM ≥ 4t</td>
<td>TSX-GL ≤ 4t; TinySTM ≥ 5t</td>
</tr>
<tr>
<td>genome</td>
<td>high (97)</td>
<td>low</td>
<td>TinySTM</td>
<td>TinySTM</td>
</tr>
<tr>
<td>yada</td>
<td>high (99)</td>
<td>medium</td>
<td>SwissTM</td>
<td>SwissTM</td>
</tr>
<tr>
<td>labyrinth</td>
<td>high (100)</td>
<td>high</td>
<td>STMs (except TL2)</td>
<td>STMs (except TL2)</td>
</tr>
</tbody>
</table>
petitive and better than the corresponding STMs, but they are far from TSX-GL in both metrics. It is worth noticing that the small and rare atomic blocks of this benchmark allow the GL approach to scale up to 3 threads. This explains the considerable success of TSX-GL in this benchmark, as a transaction that resorts to the GL is still able to run concurrently with other threads that are not under an atomic block at that time.

**SSCA2:** This benchmark shows a similar trend between TSX variants, but with the significant difference that all STM approaches scale better as the degree of parallelism increases. Here, TSX-GL is only slightly better than the best STM, and this is consistent across all the thread counts. Also interestingly, TSX-TL2 improves little and fares rather bad on the energy side. This, however, is not the case for TL2 or TSX on their own, and as such is an artefact of the hybrid implementation integration. Finally, the reduced time within atomic blocks still allows the GL approach to scale up to 2 threads, which justifies the advantage of TSX-GL. However, this effect is smaller than in Kmeans, which also matches the fact that TSX-GL achieves less improvements over other approaches.

**Intruder:** Here TSX-NOREC (and TSX-GL to some extent) are competitive and even better (until 5 threads) than the best STMs (except for TL2). Since TL2 performs poorly in this benchmark, this also drags TSX-TL2 behind in both metrics. Interestingly, both TL2 and TSX-TL2 improve slightly performance with more threads, but TL2 consumes more power whereas TSX-TL2 slightly decreases it.

**Vacation:** Once again we see that the performance of TSX-TL2 is quite disappointing, as indeed TL2 itself performs poorly in this scenario. As we shall see throughout this study, TL2 is by far the worst STM among those considered, which is a result of having a similar algorithmic and synchronization complexity to that of SwissTM and TinySTM, while detecting conflicts lazily at commit-time. This results in TL2 doing useless work more often, whereas SwissTM and TinySTM restart the speculation faster when reacting to conflicts. On the other hand, NOrec is simpler, both in algorithmic as well as synchronization terms, reducing its instrumentation overheads and maximizing its performance at low thread counts. With regard to the other approaches, TSX-GL and TSX-NOREC are competitive with STMs until 4 threads. At higher parallelism degrees, their performance degrades due to contention on L1 caches caused by hyper-threading. Analogous results are achieved for what regards power consumption. It is interesting to note that TSX-GL performs worse than TSX-NOREC at 8 threads, but the two consume approximately the same power. This is a result of the power savings that are achievable with the lock acquisition in TSX-GL.

**Genome:** In the three last benchmarks we have either transaction-heavy or high-contention workloads, characterized by large transaction foot-prints. These conditions are clearly a much more favourable playground for STMs. In this case, we see a clear (and consistent across benchmarks) distinction between TL2 and NOrec, as these two lag behind in both metrics particularly at higher thread counts. Interestingly, we can see that TSX-TL2 performs best among the TSX variants at a higher concurrency degree, which is a singularity among all benchmarks. This benchmark also shows a clear trend when the 5th thread is used: all approaches stabilize (or even decrease) performance at that point, due to hyper-threading. Interestingly, this effect is not so noticeable on the energy side, as STMs are still able to reduce the power consumption as parallelism increases. This highlights an interesting trade-off of hyper-threading: it allows sub-linear speed-ups only, but it also consumes little additional power. This fact is favourable to STMs, as TSX approaches generate more transactional aborts when hyper-threading is used, due the higher contention on the L1 cache.

**Yada:** This benchmark shows one scenario where TSX-GL performs poorly, with slowdowns above 3 threads. HyTMs follow closely their fallback STMs’ performance, as TSX is not able to succeed. This is also a case where TinySTM and SwissTM perform better than the other two STMs. This benchmark presents no surprises in the energy-efficiency, whose trends are highly correlated with the performance.

**Labyrinth:** Here we see STMs performing best and very alike each other. TSX-GL does not improve with thread count, simply because most transactions exceed the hardware cache capacity and, as such, eventually follow the fallback path which is a sequential bottleneck given the GL. For this reason, TSX-TL2 and TSX-NOREC obtain some improvements, exactly because the fallback allows for concurrency, contrarily to the global lock on TSX-GL. This scenario highlights, however, that HyTMs are capped by either TSX or the fallback STM — as such, it is dubious whether they are practical (at least when used with TSX), or if it

---

**Table 5: Transactional abort rate (%)**. For STM we show the lowest and highest values obtained (across all considered STMs).

<table>
<thead>
<tr>
<th>benchmark</th>
<th>kmeans</th>
<th>ssc2</th>
<th>intruder</th>
<th>vacation</th>
<th>genome</th>
<th>yada</th>
<th>labyrinth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 thread</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STM</td>
<td>0 - 0</td>
<td>0 - 0</td>
<td>0 - 0</td>
<td>0 - 0</td>
<td>0 - 0</td>
<td>0 - 0</td>
<td>0 - 0</td>
</tr>
<tr>
<td>TSX-GL</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>49</td>
<td>11</td>
<td>46</td>
<td>95</td>
</tr>
<tr>
<td>TSX-TL2</td>
<td>0</td>
<td>0</td>
<td>36</td>
<td>94</td>
<td>35</td>
<td>19</td>
<td>53</td>
</tr>
<tr>
<td>TSX-NOREC</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>40</td>
<td>6</td>
<td>19</td>
<td>53</td>
</tr>
<tr>
<td>4 threads</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STM</td>
<td>10 - 34</td>
<td>0 - 0</td>
<td>0 - 0</td>
<td>0 - 6</td>
<td>1 - 51</td>
<td>5 - 58</td>
<td>4 - 13</td>
</tr>
<tr>
<td>TSX-GL</td>
<td>26</td>
<td>0</td>
<td>22</td>
<td>69</td>
<td>31</td>
<td>48</td>
<td>100</td>
</tr>
<tr>
<td>TSX-TL2</td>
<td>50</td>
<td>74</td>
<td>74</td>
<td>100</td>
<td>45</td>
<td>84</td>
<td>60</td>
</tr>
<tr>
<td>TSX-NOREC</td>
<td>31</td>
<td>46</td>
<td>29</td>
<td>66</td>
<td>17</td>
<td>31</td>
<td>55</td>
</tr>
<tr>
<td>8 threads</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STM</td>
<td>25 - 54</td>
<td>0 - 0</td>
<td>3 - 57</td>
<td>0 - 10</td>
<td>0 - 1</td>
<td>7 - 60</td>
<td>8 - 23</td>
</tr>
<tr>
<td>TSX-GL</td>
<td>42</td>
<td>1</td>
<td>33</td>
<td>72</td>
<td>48</td>
<td>47</td>
<td>100</td>
</tr>
<tr>
<td>TSX-TL2</td>
<td>60</td>
<td>99</td>
<td>92</td>
<td>100</td>
<td>53</td>
<td>92</td>
<td>69</td>
</tr>
<tr>
<td>TSX-NOREC</td>
<td>44</td>
<td>88</td>
<td>62</td>
<td>99</td>
<td>69</td>
<td>39</td>
<td>60</td>
</tr>
</tbody>
</table>
would be preferable to adaptively employ the most promising technique (TSX-GL or an STM) based on the workload.

6.2 Insights on TM Efficiency

In this section we shed some additional light on the factors dictating the trends observed in the experiments. To this end, in Table 5 we report the average abort rate across benchmarks and threads for each speculative mechanism. This represents the percentage of speculations that do not complete. Since there are four STMs under evaluation, we show the minimum and maximum abort rates among them — typically the smallest abort rate belongs to TinySTM and SwissTM, whereas TL2 yields the maximum abort rate.

Once again, we structure the table considering the different categories of workloads. As we move right (more contention or transaction-intensive workloads) and down (higher degree of parallelism), TSX approaches increase abort rates, which causes the loss of efficiency shown in the previous section. These results highlight that TSX has non-negligible aborts in many occasions where STMs abort very little.

In Fig. 2 we consider four different benchmarks, representative of scenarios that allow to derive insights on the efficiency of the considered TSX variants. In those plots we present a breakdown of the reasons motivating transactional aborts, for each TSX mechanism. We distinguish aborts caused by exceeding the capacity of L1 cache; micro-
architectural instructions or states forbidden by TSX, such as some system calls; data contention resulting in conflicts; and interaction between TSX and the fallback paths, such as checking if the GL is free in TSX-GL or more complex logic in the case of HyTMs.

Kmeans’ breakdown shows that, as expected, as concurrency increases, also abort rates increase due mainly to data conflicts. It is worth mentioning that Kmeans is the benchmark with the least average aborts for TSX variants. Half of the aborts are due to conflicts, whereas the rest is motivated by a non-negligible percentage of aborts due to architectural instructions. This is something intrinsic to TSX, which is common throughout different benchmarks. The fact that these aborts occur less often in this benchmark allows TSX to obtain the most favourable results among all benchmarks.

In Yada and Labyrinth, instead, the workloads are much more transaction-intensive with non-negligible conflict rates. On top of this, the capacity of L1 caches is often exceeded by the hardware transactions (this is particularly visible in Labyrinth, where this phenomena dominates the aborts). This explains why the TSX variants followed up closely the performance of their fallbacks (with some constant overhead). HyTMs have a reduced abort rate because the fallback’s software transactions are also taken into account in these statistics, on top of the hardware transactions — since software transactions have little aborts due to the uncontended workload, they amortize the overall abort rate. In TSX-GL, instead, the fallback executes non-speculatively due to the global lock, so we only count statistics for the hardware transactions there.

Finally, SSCA2 shows a completely different scenario, in which TSX-GL generates almost no aborts (in line with STMs’ behaviour), whereas HyTMs have enormous abort rates, dominated by the interaction with the fallback path.

This motivates to better understand the usage of the fallback path in the HyTMs. Table 6 shows the percentage of transactions that were executed in the fallback (i.e., not purely in hardware). We also show the percentage of transactions in the fallback that are able to execute in a fast mode, i.e., a mode in which the transaction executes in software but the commit is boosted by using a reduced hardware transaction [30] (as explained in Section 3). For every table cell we show the percentage corresponding to 1 and 8 threads. Overall, the percentages vary linearly from 1 to 8 threads, for which reason we omit the intermediate values.

We start by highlighting in SSCA2 how both HyTMs are able to execute purely in hardware with 1 thread (they trigger the fallback < 1% of the transactions). However, a higher thread count typically results in executing in the fallback mode almost all the time, which matches the idea conveyed by Fig. 2(d). In particular, for this benchmark, the ability to rely on hardware to speed up the software fallback path is reduced from above 90% to 14% or even less.

These results for HyTMs show that TSX-TL2 triggers the fallback more often, and is able to execute in the fast mode less frequently than TSX-NOrec. This justifies the advantage of TSX-NOrec, which fared better across all benchmarks in Section 6: TSX-NOrec executes the fallback software transactions in fast mode for most of the time. The reason is that the much simpler design of NOrec allows for a much easier integration with TSX in a HyTM.

Ideally one may want to also rely on more scalable STMs, like TinySTM or SwissTM, in the fallback of TSX. However, due to the higher complexity of their algorithms, coupling them efficiently with HTM is a challenging task, and, in fact, we are not aware of any proposal in this sense in literature.

Finally, it has been pointed out in [8] that, in order to support efficient HyTMs, it is desirable to have hardware support for selective non-transactional memory accesses in the scope of transactions. Such a feature is not currently supported in TSX, whereas its inclusion was, e.g., planned in AMD’s HTM proposal [6] (which was never commercialized). Hence, an interesting research direction suggested by this study is to investigate the impact of supporting non-transactional accesses, not only in terms of performance and energy, but also in terms of architectural intrusiveness.

![Figure 2: Breakdown of reasons causing transactional aborts for TSX variants.](image-url)

Table 6: Rate (%) of triggering the fallback on HyTMs and of executing it in fast mode. Intervals of values are shown, ranging from 1 (lower) to 8 threads (upper bound).

<table>
<thead>
<tr>
<th></th>
<th>TSX-TL2</th>
<th></th>
<th>TSX-NOrec</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fallback</td>
<td>Fast</td>
<td>fallback</td>
<td>Fast</td>
</tr>
<tr>
<td>kmeans</td>
<td>&lt; 1 - 77</td>
<td>92 - 32</td>
<td>&lt; 1 - 78</td>
<td>100 - 24</td>
</tr>
<tr>
<td>ssca2</td>
<td>&lt; 1 - 99</td>
<td>91 - 2</td>
<td>&lt; 1 - 86</td>
<td>95 - 14</td>
</tr>
<tr>
<td>intruder</td>
<td>33 - 88</td>
<td>98 - 39</td>
<td>3 - 55</td>
<td>100 - 52</td>
</tr>
<tr>
<td>vacation</td>
<td>94 - 100</td>
<td>43 - 3</td>
<td>38 - 99</td>
<td>100 - 89</td>
</tr>
<tr>
<td>genome</td>
<td>90 - 100</td>
<td>97 - 71</td>
<td>6 - 67</td>
<td>100 - 91</td>
</tr>
<tr>
<td>yada</td>
<td>18 - 78</td>
<td>50 - 34</td>
<td>17 - 32</td>
<td>99 - 82</td>
</tr>
<tr>
<td>labyrinth</td>
<td>58 - 100</td>
<td>14 - 2</td>
<td>54 - 98</td>
<td>10 - 3</td>
</tr>
</tbody>
</table>
7. BENCHMARKS USING FINE-GRAINED LOCKING

Most of the STAMP benchmarks have an irregular nature, which makes it very challenging to derive fine-grained locking approaches. We start, in Section 7.1, by focusing on a subset of three STAMP benchmarks, for which we could craft an ad-hoc fine-grained locking strategy. We then present results for Memcached in Section 7.2 and for two concurrent data structures in Section 7.3.

7.1 Fine-grained Locking in STAMP

As already mentioned, implementing a fine-grained locking strategy is a complex task for most of the STAMP benchmarks. We were, however, able to devise fine-grained locking strategies for three of the STAMP benchmarks, whose results we report in Fig. 3. Besides fine-grained locks (FL), we also show results for TSX-FL, which combines hardware transactions with a fallback path that relies on FL. Naturally, the combination of both schemes in TSX-FL requires hardware transactions to read all necessary locks as being free. We then compare these two approaches with TSX-GL and TinySTM, which were the best mechanisms in our previous experiments, and remove the others to improve the readability of the plots.

TSX-FL presents one advantage over TSX-GL, in that the fallback path allows for threads to proceed in parallel if they require different locks (which is highly likely if there is little data contention). However, this has the drawback that more locks have to be checked (during speculative executions) or acquired (during the fallback executions). Hence, there is a clear trade-off that is subtle and difficult to manage.

Recall that Kmeans and SSCA2 were the two benchmarks with workload characteristics more amenable to TSX-GL. This is justified by the low frequency of activation of the fallback path. As such, TSX-GL incurs minimal overhead thanks to the hardware speculation and to the avoidance of any software-based instrumentation. Therefore, it is not a surprise that fine-grained locking is of no advantage in this scenario: each lock acquisition represents a synchronization point, whereas for TSX-GL there exists only explicit synchronization at the hardware level when a transaction attempts to commit. Note, however, that FL is consistently better than the best STM (TinySTM). This fact is even more relevant from the energy perspective, where the gap between FL and TinySTM is larger. Since the TSX fallback is not triggered often, then TSX-FL goes through the additional verifications over more locks that are useless most of the time (to ensure a correct integration of the fallback with hardware transactions), which explains its lower performance in this kind of workload.

In SSCA2 we see a different behaviour as both TSX variants perform quite similarly. This is explained by the fact that the fine-grained scheme is not very efficient: its locks are relatively coarse, which induces unnecessary serialization. This has the side-effect of making TSX-FL competitive with TSX-GL, because both have a similar effort in checking the locks in the speculative executions to ensure correct integration with the fallback. Notice how the FL scheme still performs better than GL, which is a consequence of the higher degrees of parallelism achievable by reducing lock granularity. This confirms an expectable trade-off concerning lock granularity: the more fine-grained, the best the fallback performs; however this can have an impact on the performance of the speculative executions as we saw for Kmeans.

Finally, Intruder spends a large fraction of time within atomic blocks. As already discussed, this workload is more advantageous for STMs than for TSX. It is not surprising to see that TSX-GL is no longer the most competitive choice (although it still fares best until 3 threads). The interesting fact is that this kind of workload is more beneficial for FL. With more threads, TinySTM degrades its scalability, and is surpassed by FL. From an energy perspective, it is even clearer that FL is the best choice comparing to TinySTM, as it is almost always consuming less energy. TSX-FL suffers from the overheads of checking additional locks, until 3 threads, for which reason it is not as good as TSX-GL. However, at that point TSX triggers the fallback more often, which justifies the use of fine locks and allows TSX-FL to perform substantially better than TSX-GL. On the energy side, TSX-FL is closer to TinySTM, following the trend of STMs that often fare worse on the energy side to obtain comparative levels of performance to the other approaches.
7.2 Memcached

Memcached is a popular distributed object caching system [31]. In this study, we rely on a recent TM-based porting [35], and use the original Memcached as the basis for FL. We used the memslap tool, configuring the workload with 95% gets and 5% puts, 8 threads and a concurrency of 256.

In Memcached it is not really possible to measure, for reference purposes, the performance of a sequential execution, because there is always concurrency due to the existence of a pool of maintenance threads. Hence, we present the peak throughput obtained using the maximum number of available hardware threads (see Fig. 3(h)). The results show that FL has the best performance, but TSX-GL is only 7% behind. This is a significant achievement as the effort to devise such fine-grained locks is considerably higher than using TSX-FL. Also, since FL is quite optimized, it is expectable that TSX-FL is not able to extract any further parallelism. Interestingly, with this benchmark, TinySTM is not competitive because the instrumentation overheads are amplified by the short and uncontented transactions.

7.3 Concurrent Data Structures

We now consider two concurrent data structures, namely a red-black tree and a hashmap, which represent particularly relevant use cases for TM given the complexity of designing efficient fine-grained locking strategies for these scenarios.

Fig. 4 shows two different scenarios: we consider a small hashmap (512 buckets) with only 10% transactions performing writes (the rest are lookup operations), and a large red-black tree (1 million items) with 90% transactions performing updates. In the former case, TSX-GL achieves perfect linear scalability, which is a consequence of its negligible overheads and of the very reduced abort rate. With larger transactions, the gains achievable by TSX tend to diminish, although it still remains a very competitive solutions.

Table 4(e) shows a spectrum of workloads in red-black tree, by considering the normalized EDP of TSX-GL against the best alternative in each experiment. For this, we vary the size of the tree and the percentage of write transactions. The trend is clear in this table: TSX behaves best with light workloads, and looses advantage when transactions become larger or write-intensive. This confirms the results of the analysis that we performed for STAMP, given that, also in this case, TSX shines most when atomic blocks have little duration and the workload is not fully transactional.

8. RESEARCH DIRECTIONS SUGGESTED BY OUR STUDY

We now identify some relevant research directions that emerged from the analysis of our experimental study:

- The overall performance of the tested HyTM solutions is quite disappointing. These findings contradict the simulation results published in several previous works, e.g., [30]. Our analysis suggest that the root cause of the problem is related to the inefficiency of the mechanisms used to couple hardware and software transactions, which is generating a large number of spurious aborts. However, further research is due in order to understand what can be done to address such a problem. An interesting research question, in this sense, is whether the availability of support for enabling non-transactional memory accesses while executing hardware-assisted atomic blocks could indeed allow for more efficient interplay between HTM and STM (which has been assumed by other works in the area of HyTM, e.g. [33]). A related research question is how to support such a feature while minimizing the disruptiveness of the changes required at the hardware level—an aspect that cannot be overlooked given the complexity of modern processor architectures.

- As mentioned in Section 5.3, the performance of TSX is significantly affected by the retry policy (e.g., the settings of the number of retries upon abort, and the choice of how to react to capacity aborts). While in our study we used the configuration that performed best on average, as shown in Table 7, significant speedups (up to 80%) with regard to the configuration used in our study can be achieved by ad-hoc tuning the retry policy for the specific workload—even more could be achieved by considering the specific concurrency degree as well. Unfortunately, this is a tedious and error prone task that is not desirable to delegate to programmers. Hence, these findings highlight the relevance of devising solutions for adaptively tuning these parameters in an automated manner. The key challenge is how to do it with minimal overhead, given that the cost imposed by self-tuning approaches targeting STMs (based on complex machine-learning [34] or analytical models [11]) is going to be strongly amplified in HTM settings because there exists no instrumentation as in STMs. In the light of these findings, we have concurrently obtained some initial results with regard to this research direction [15].

- Our study has used (selective) manual instrumentation when considering both STMs and HyTMs, i.e. only the relevant subset of memory locations accessed in atomic blocks have been traced. As an alternative, one could rely on the compiler to automatically instrument atomic blocks with calls to the TM runtime. The plots in Fig. 5, which were

![Figure 4: Data Structures varying contention level.](image-url)
obtained using the C++ TM extension integrated in GCC 4.8.2, show that non-selective instrumentations can impact performance by approximately 20% when using TinySTM. This is a consequence of the increase of the transaction footprint (up to 3x larger with SSCA2) caused by the “blind” instrumentation performed by GCC.

Not only these results unveil the possibility of optimizations in existing compiler’s support for STM, but also provide an additional compelling motivation to incorporate support for selective instrumentation in HTM. Indeed, we have shown that capacity exceptions are one of the key sources of aborts with HTM. Hence, techniques capable of achieving noticeable reductions of the transactions’ footprint are expected to strongly benefit HTM’s performance. These considerations open interesting research avenues investigating cross-layer mechanisms operating at the compiler and architectural level, and aimed at supporting selective instrumentation in a way that is both convenient for the programmer (i.e., possibly fully transparent) and sufficiently non-intrusive to simplify integration in existing architectures.

9. CONCLUSIONS

This paper analyzed extensively the performance and energy efficiency of several state of the art TM systems. We compared different TM solutions (software, hardware and combinations thereof) among each other and against lock-based systems. Our study demonstrates that the recent HTM implementation by Intel can strongly outperform any other synchronization alternative in a set of relevant workloads. On the other hand, it also identified some critical limitations of Intel TSX, and highlighted the robustness of state of the art STMs. These software implementations achieve performance competitive with fine-grained locking, and outperform HTM in workloads encompassing long and contention-prone transactions.

Furthermore we have shown that the performance of Hybrid TM, when used in combination with TSX, is normally quite disappointing; we determined that the root cause of this surprising result lies in the inefficiency of the mechanisms used to couple software and hardware transactions. Finally, our study allowed to identify a set of compelling research questions, which, we believe, should be timely addressed to increase the chances of turning HTM into a mainstream paradigm for parallel programming.

Acknowledgements: We thank Konrad Lai, Ravi Rajwar and Richard Yoo from Intel for the insightful discussions about TSX during the conduct of this research.

This work was supported by funds from Fundação para a Ciência e Tecnologia under PEst-OE/EEI/LA0021/2013 and by GreenTM EXPL/EEI-ESS/0361/2013.

10. REFERENCES


