COMP60621

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: Tuesday 19th January 2016

Time: 14:00 - 17:00

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.

[PTO]

1. Provide an analysis of **one** of the following two papers**:**

a) A) Flask Coherence: A Morphable Hybrid Coherence Protocol to Balance Energy, Performance and Scalability HPCA 2015

or

b) B) Transactionalizing Legacy Code: an Experience Report Using GCC and Memcached ASPLOS 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)

END OF EXAMINATION

Flask Coherence: A Morphable Hybrid Coherence Protocol to Balance Energy, Performance and Scalability

Lucia G. Menezo Valentin Puente Jose-Angel Gregorio
University of Cantabria
Santander, Spain
{gregoriol, vpuente, monaster}@unican.es

Abstract—**This work proposes a mechanism to hybridize the benefits of snoop-based and directory-based coherence protocols in a single construct. A non-inclusive sparse-directory is used to minimize energy requirements and guarantee scalability. Directory entries will be used only by the most actively shared blocks. To preserve system correctness token counting is used. Additionally, each directory entry is augmented with a counting bloom filter that suppresses most unnecessary on-chip and off-chip requests. Combining all these elements, the proposal, with a low storage overhead, is able to suppress most traffic inherent to snoop-based protocols. With a directory capable of tracking just 40% of the blocks kept in private caches, this coherence protocol is able to match the performance and energy of a sparse-directory capable of tracking 160% of the blocks. Using the same configuration, it can outperform the performance and on-chip memory hierarchy energy of a broadcast-based coherence protocol such as Token by 10% and 20% respectively.**

To achieve these results, the proposal uses an improved counting bloom filter, which provides twice the space efficiency of a conventional one with similar implementation cost. This filter also enables the coherence controller storage used to track shared blocks and filter private block misses to change dynamically according to the data-sharing properties of the application. With only 5% of tracked private cache entries, the average performance degradation of this construct is less than 8% compared to a 160% over-provisioned sparse-directory.

Keywords – coherence protocol; multi-core; CMPs

I. INTRODUCTION

The enforcement of hardware coherence in contemporary CMPs with complex on-chip cache hierarchy constitutes an interesting problem of competing trade-offs in cost, energy and performance. The solutions adopted for this problem vary greatly from system to system, it being unclear whether there is a universal solution. To achieve the expected scalability of future many-core CMPs, the use of a directory-based coherence protocol seems unavoidable. In contrast, current high-performance commercial systems seem to favor the use of broadcast-based coherence protocols [9], [14], [16]. This approach is used sometimes for the intra-chip and off-chip realm even with a large number of coherent cores. Consequently, although from the energy standpoint, broadcast-based coherence loses its appeal when the number of cores in the system grows, its performance advantage and complexity compared to directory-based coherence makes it predominant.

This work was supported by the Spanish Government, under contract TIN2013-43228-P, the University of Cantabria, under grant VP07, and by the HiPEAC European Network of Excellence.

In general, directory-based approaches demand inclusivity to guarantee correctness. When the number of cores grows, simple approaches such as duplicate-tag directories become inefficient due to the large associativity required. To meet energy constraints, the solution adopted is to over-provision the directory to minimize the unnecessary evictions in private caches due to directory conflicts under a constrained (and realistic) associativity [15]. Unfortunately, the size of the private caches is growing as a consequence of the larger and larger sizes in the last-level cache (LLC), which implies that the number of blocks the directory must track has to be larger.

On the other hand, broadcast-based coherence protocols interrogate all coherence agents in the chip when a core misses the desired block in its private caches. In order to guarantee correctness, neither inclusivity nor additional structures to track block copies are required. Therefore, resource utilization is better. Nevertheless, this is achieved at the cost of increasing the traffic and cache snoops, which will decrease the energy efficiency of the system. In small-scale systems, this might be tolerable, but when the size of the system grows, the impact will be noticeable, and possibly unsustainable. A more subtle effect, but a no less relevant one, is the on-chip resource contention that characterizes these protocols. As a result, on-chip access latency can be affected, perhaps degrading the CMP performance under some particular usage scenarios.

From this standpoint, it would appear that a pure coherence protocol might not be the most suitable approach to tackle the problem. Intuitively, it seems that the coherence protocol should somehow hybridize the best of both types: trying to attain the performance effectiveness and implementation cost of a broadcast-based coherence protocol with the energy efficiency of a directory-based one. This paper addresses this task and successfully attains a new coherence protocol, denoted FLASK (FiLtered and Allocated just by Shared block Keeper) coherence, which can scale as a directory-based coherence protocol does, while achieving cache effectiveness similar to a broadcast-based one.

FLASK combines three components in a single logic substrate. It uses a directory to track blocks that are *actively shared*. Therefore, private blocks, which are the most frequent case, never allocate an entry in the directory unless they are accessed during their cache lifetime (i.e., from miss to eviction) by another core. Moreover, the inclusiveness property (both for

directory and LLC) is not required, avoiding external private cache invalidations due to directory conflicts. Correctness is guaranteed through token counting. When the copies of a shared block are not being tracked by the directory, after a new request to the coherence controller a broadcast to all coherence agents is generated. The replies are used to reconstruct the directory entry. This approach to perform reconstruction of directory entries on demand was introduced by MOSAIC [27]. Nevertheless, MOSAIC allocates directory entries for any on-chip miss (i.e., for both private and shared blocks) and always generates a broadcast if there is a miss in the directory. To minimize unnecessary snoops, each entry in the directory is assisted by a filter that suppresses most unnecessary snoops. This component can directly identify the majority of on-chip misses. As a consequence, nearly all of the requests to off-chip coherence agents (memory controllers and/or off-chip coherence fabric) are not delayed. In the least common case, misses in a private cache of an actively shared block are always tracked by the filter and dealt with through a multicast to the on-chip coherence agents (private cache coherence controllers). The filter also has to avoid unnecessary off-chip requests. Finally, the architecture of the filter proposed allows us to dynamically assign, according to the sharing degree of the running workload, storage capacity in the coherence controller either to track shared blocks in the directory or to identify privately held blocks. This construct allows us to reduce the size of the directory even further.

The main contributions of the paper are:

- The hybridization of directory and a broadcast coherence in a unified logic substrate with optimized implementation and energy costs.
- The proposed strategy can achieve the performance of a conventional over-provisioned sparse directory, while tracking less than 40% of the private cache entries. Similarly, it improves on Token coherence protocol performance by 10% and energy delay product by 20%.
- With only 5% of tracked private cache entries, average performance degradation is less than 8% with respect to a 160% over-provisioned sparse-directory.
- We show that, using an adaptive storage assignation at the coherence controller according to the workload properties, we can reduce even further the resources of the directory. This is based on the distinctive properties of the filter mechanism used. Under these circumstances, the proposal is able to match a sparse-directory performance while tracking only 20% of private cache entries.

II. BACKGROUND AND MOTIVATION

A. Directory Coherence Shortcomings

Directory-based protocols seem to be an attractive approach to enforce cache coherence in a CMP. Nevertheless, when the number of cores is high and the on-chip hierarchy complexity grows, the directory is difficult to scale. We will assume a multilevel hierarchy with a shared last-level cache (LLC). A cost effective way to track the coherence information in this structure is to use a sparse-directory [15]. In contrast to in-cache directory, only the data actively used by the cores (i.e. contained in private caches) has to be tracked. This is much more cost effective, since the storage devoted to LLC is usually larger than private caches. Additionally, LLC has to be banked in order to alleviate access contention. In most cases, it is appealing to use scalable interconnects to connect these banks [17]. Under these circumstances, it is straightforward to bank the directory accordingly and attain an easy-to-handle, distributed structure.

In an on-chip cache, similar to state-of-the-art systems [10], [14], [16], in order to close the gap in the access time between a small L1 (dominated by processor clock cycle) and a very large LLC (dominated by main memory access time), an intermediate level is required. As a consequence the number of blocks that the directory has to track is larger. Additionally, those intermediate levels usually have a substantial associativity. Recent designs [14][16] also require a large associativity for L1. In summary, the number of blocks that can be mapped in a set of the directory could be high. Although in some early CMPs [21] the directory is provisioned to keep all the blocks in the private caches, when the number of cores or private cache complexity and size grows, this is not feasible due to the enormous associativity required by the directory. However, reducing this associativity increases the eviction of blocks in the private caches due to conflicts in the directory.

A rule of thumb [15] suggests that over-provisioning the directory with twice the capacity required to track the private caches will diminish the problem. Since the amount of cache to track and the sharing vector will increase with the number of cores, the cost of the directory grows quadratically. Otherwise, the number of invalidations in the private caches would grow significantly [12], impairing system performance.

Fortunately, the directory cost problem can be tackled by considering the application semantics. It is known that most memory regions are accessed privately by a single core most of the time [11][3]. If the directory is aware (actively [11][3] or passively [27][12]), we can reduce the number of private blocks that we have to track in the directory. Consequently, we can reduce the number of entries without interfering with the private cache performance.

B. Broadcast Coherence Shortcomings

The main problem with these protocols is their scalability issues, due to the extra traffic and cache snoops that each private cache miss triggers. Similarly to the directory, this might grow when the number of cores is increased (because more traffic is required) and the private caches are bigger (because the tag snoops are more costly). With a restricted number of cores, however, broadcast seems to be the most suitable choice, bearing in mind that many commercial high-performance processors use it [10][14][16]. For these systems, cost constraints support this design decision but it might not be sustainable in future designs.

One way to tackle the problem is to use suitable interconnections that minimize the utilization of the same resource in the network by copies of the same message. This will be done by supporting on-network broadcast and/or on-network gather [19][22]. At the same time, to avoid both communication and tag snoop overheads, many works advocate filtering [10][28] or adapting the protocol behavior to the bandwidth availability [25][30]. In order to filter out unnecessary memory

controller (MC) accesses or off-chip coherence fabric interfaces (XC), additional mechanisms should be provided [16][10].

C. Cache Coherence Hybridization

Many of the previous solutions proposed to alleviate both directory and broadcast limitations are based on a complementary design alternative. For example, in a directory for shared blocks, the approach followed is to snoop all or a subset of the coherence agents to see whether they have a copy of the block when a request misses in the directory and/or LLC [12][27]. In other works, the coherence protocol acts as a snoop-based protocol does. Therefore, the resulting protocol mimics a directory protocol in some cases and a broadcast protocol in others. The same observation could be made for some broadcast-based coherence protocols, which reduce the energy overheads through the insertion of structures to filter unnecessary cache snoops [28]. In some way, most of these solutions use a "base" approach (either directory or broadcast) and use the complementary one to compensate for its inherent limitations. Similarly, our proposal combines both strategies: a standard sparse directory [15] and token coherence [24]. We will use a directory-like structure to track most shared blocks (with precision in a sharing vector) and private block presence (approximately). In both cases, token coherence is used to discover when, after a miss, a block should be classified in one or other group. The information present in this structure will be used to minimize the on-chip and off-chip traffic. Then, intuitively, we can say that both facets of the coherence protocol operate with similar levels of relevance. Our objective is a mechanism able to get the best of both worlds: the performance of a broadcast-based protocol with the energy efficiency of a directory-based protocol, while incurring a minimal storage overhead.

III. FLASK COHERENCE

The coherence controller is composed of two key elements: a sparse directory that will be used only for actively shared data and a filter that will be in charge of determining when, for a given address, there is a copy of the block in any private cache. In both cases, we will use token counting [24] with two objectives: guaranteeing that *coherence invariants* are respected and monitoring when the filter should be updated. The number of tokens of a block that enters the chip is set to the number of cores. Then, a copy of a block can be read if it has at least one token. A copy of the block can be written if it has all the initial tokens. In this way, we guarantee single writer/multiple readers invariant for each block. Token counting will be useful to discover whether a block is being actively shared or not and to act accordingly, not just in the sparse directory but also in the filter. Filter management is mostly isolated from coherence protocol. In fact, the coherence controller will act correctly with an empty filter. Token counting allows us to use this strategy: it helps the filter to determine when the last copy of the block leaves or enters the chip. The filter should be understood as a traffic filtering device for both on-chip and off-chip traffic. A high-level representation of the coherence controller is shown in fig. 1. Next, we will detail how each element operates.

A. Sparse Directory (for the majority of the actively Shared Blocks)

Broadcast Token Coherence (TokenB) resolves misses in the private cache by issuing a request to all the potential coherence agents in the chip where a copy can reside. To avoid this scenario, we use a sparse-like directory. In contrast to conventional sparse directories, this will track only actively shared blocks. This means that when a block is missing in the chip, an entry will be allocated only in the corresponding private cache. If the block remains private, when it gets replaced, it will be progressively moved to further levels until it gets evicted to the LLC, which is non inclusive. We denote the time between these two points the *private caching period*. The block is actively shared if during the private caching period it is accessed by another core in the system. If this circumstance arises, we need to allocate an entry in the directory with its sharing information, i.e. with the two cores if the second request was a read, or with the last core if it was a write.

Thus when a processor misses in the corresponding private cache, it is not possible to determine whether there are copies in other private caches just by checking whether there is an entry allocated in the directory or not. Note that when a directory entry is evicted no external invalidations are sent to the private caches that hold a copy of the block. At first, after any miss in the directory, we will assume that it will initiate a reconstruction

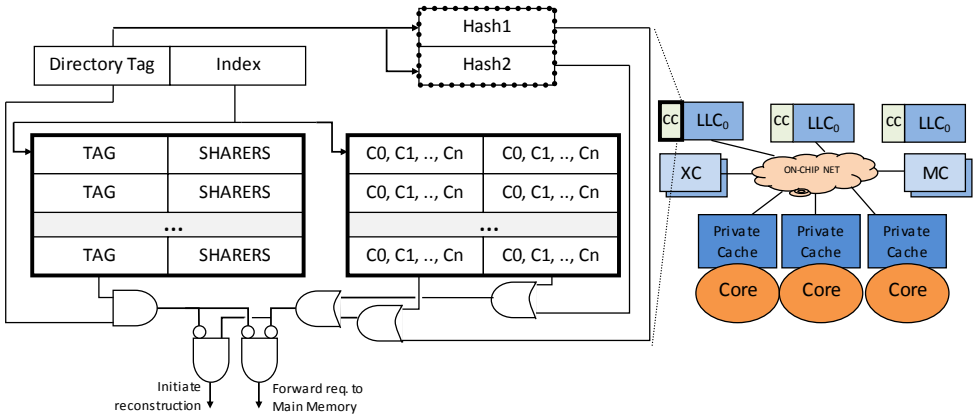

Fig. 1. High-level representation of a lookup in the FLASK coherence controller (simplified vision with a direct-mapped sparse directory and a parallel counting bloom filter with n counters and two hash functions).

process, which will snoop the remaining coherence agents in order to recreate the sharing vector. Using token counting, the directory will be aware when the information received is enough to unblock the pending memory operation. The on-chip coherence agents will respond with the count of tokens owned for the requested data.

There will be 3 potential scenarios:

(1st Case) If the response is delivered from memory, we consider that the block is private and the requesting core will be the owner of the data. Then the directory can directly forward the data and tokens to the requesting processor without allocating an entry in either the directory or the LLC (i.e. a private block will not evict the information corresponding to another shared block).

(2nd Case) If the response comes from LLC and it indicates that LLC has all the tokens, the directory knows that there is no other copy of the block in the chip and it can instruct the LLC to forward the block to the requestor processor. In this case, like in memory responses, we know that the block is not actively shared and so it will not be necessary to allocate a new entry in the directory.

(3rd Case) If any response comes from any of the cores, i.e. from their private caches, we know that the block is being actively shared and so a directory entry will be allocated. If the pending operation is a *load*, the directory instructs the private cache with the owner token to forward the data to the requestor and it keeps counting incoming answers. When the location of all the tokens is known, the sharer vector is accurate. If the pending operation is a *store*, the directory instructs all the private caches with tokens to forward them to the requestor core (consequently invalidating the data) and also requests the cache where the owner token is allocated, to forward the data block to the requestor processor. When all the tokens are received by the requesting core, it knows that it can proceed with the operation, and unblock the directory entry.

The reconstruction approach is based on the one proposed in MOSAIC [27]. Nevertheless, FLASK never allocates an entry in the directory unnecessarily (i.e. for a private block). Eventually, MOSAIC can use the directory to allocate mostly shared blocks, but on-chip misses will trigger directory replacements. However, in any case, this will imply a directory-induced private replacement like in conventional sparse directories. It will increase the effort necessary to reconstruct sharing information if the replaced entry was shared through a multicast operation. Like in the Stash Directory [12], we will allocate only shared data. In contrast to our proposal, the Stash directory requires an inclusive LLC to identify shared blocks. This means that in FLASK (or MOSAIC) more blocks missed in private caches will be located in LLC because of their greater effective capacity. In fact, the 2nd scenario will be the most common case. Requesting the block first from the LLC, in most cases, will make it unnecessary to ask the remaining coherence agents. Finally, like in MOSAIC and in contrast to the Stash directory, the eviction of an entry in the directory does not require invalidating the copies in the private caches. This will be helpful for frequent read-only blocks (such as instructions) since eventually they will be shared without using an entry in the directory. Note that many workloads, such as most server/transactional applications, have a large instruction footprint with a high sharing degree [5]. Finally, FLASK will filter unnecessary on-chip and off-chip traffic through the mechanism described next.

B. Counting Filter (for all the on-chip blocks)

Using the previously described policy to manage the directory will substantially increase the on-chip and off-chip request traffic. The second core accessing an actively shared block will be unable to determine whether it is present in the chip. Therefore, all the coherence agents should be interrogated when a miss occurs in the directory. Since our intention is to keep track of only the actively shared blocks in the directory (with the object of reducing the size), this will be the common case.

To tackle this issue, at the directory level, we have to track whether the data is present or not in the chip. Note that the aggregate private blocks tracked by this structure can be large, since both the number of cores in the chip and the private cache assigned to each one can be large. In contrast to directory, we can trade off inaccuracy in this structure in order to maintain the implementation cost constrained. This structure should also be useful to filter unnecessary memory snoops (i.e. snoop memory when there is a copy of the block present in the chip).

1) Filtering the On-chip traffic

We propose using a small counting bloom filter [6] attached to each directory entry to track all the tags contained in the private sets that map onto the entries of the directory. A counting bloom filter is an efficient, approximate set membership check that enables the tracking of a tag's presence in the private caches at a fraction of the regular cost. Since this is a filtering structure and not a correctness construction, we can tolerate the presence of false positives. These situations might increase traffic and delay memory accesses but they will not affect system correctness.

In our case, we need to increment and decrement the counters of the filter each time a block arrives or leaves the chip. The event required that triggers the increment in the counters will be an on-chip miss. This situation can be identified when the data arrives at the chip after a miss. On the contrary, decrementing each counter might be a significantly harder task. It requires identifying when there is no longer a copy of the block in the chip. We will use token counting for this purpose. As in the first scenario, if on LLC eviction a block has all the tokens, we know that there will be no other copies in the chip and consequently we can decrement the filter directly. The remaining scenario, i.e. a block that has to be evicted from LLC without all the tokens, can be much more complex to handle. Fortunately, when a block is evicted from LLC, in most cases (>99.99%) the block has all the tokens. This is an expected behavior, since the reuse distance supported by private caches is much shorter than that supported by the LLC, given the size ratios between the two caches. Instead of dealing with the problem through coherence protocol constraints, we opt to use the fact that the second case is very unlikely. Then, if we have to evict a block in LLC without all the tokens, we invalidate any private copy of the block through a broadcast. When all the tokens are collected in the LLC, we know that we can decrement the corresponding counters in the filter.

Finally, as with any counting bloom filter, the counters can overflow [4]. This situation might affect system correctness since there could be false negatives. Memory requests are likely for blocks potentially modified in the chip. The coherence protocol should contemplate this event and handle it accordingly. As this is a very unusual case, again, it is not cost effective to increase the complexity of the coherence protocol because of it. We opt for avoiding counter saturation through invalidation. If after an on-chip miss (and subsequent addition), some of the counters reach the maximum value, we invalidate all the private cache sets mapped onto that portion of the filter. Although at first sight this might seem to have a huge performance impact, in practice, with a large variety of configurations and workloads, it is extremely unlikely that this event would ever happen. This is consistent with the fact that the probability of having a counter overflow, is very unlikely [4].

2) Filtering the Off-chip traffic

Broadcast-based coherence protocols in CMP might increase the demands on the off-chip coherence agents as well as the energy overhead in unnecessary private cache tag snoops and traffic. Since this might affect a scarce resource such as the off-chip bandwidth, it is often managed by complex structures in the memory controller in charge of filtering or canceling unnecessary memory requests on the fly [10]. From our perspective, we can reuse the previously described on-chip filtering strategy to carry out this task seamlessly. When a tag is not found in the filter, as false negatives are not tolerated, we know that it will be found in memory. These are compulsory accesses. However, if there is a hit in the filter, the block will most often be in the chip, it being unnecessary to interrogate the memory. The problem we face is the false positives, which have a very low but non-zero probability. If we proceed as usual without asking the memory, the system might reach starvation. Instead of contemplating these events (e.g. through starvation detection [24]), we prevent this situation by forcing the cores without tokens to acknowledge directory entry reconstructions. In contrast to other protocols [10], the overhead of these acknowledgements is only present when there is a hit in the filter. The directory coherence controller will wait for all these answers before sending a request to the memory. Consequently, on-chip misses in which the counting bloom filter returns a false positive will be delayed until the coherence controller is aware that this block is not inside the chip. Fortunately, if the counting bloom filter is correctly dimensioned, the probability of having these false positives will be small enough to not interfere noticeably with system performance. The consequence is that memory accesses, in the worst case, will be delayed by a few tens of cycles. The positive effect of this approach is that the memory controllers do not have to be able to filter or cancel unnecessary memory requests. Note that the cost of this functionality might be substantial.

IV. Filter Design Considerations

A. Implementation: Improving Counting Bloom Filter Storage Efficiency

The storage capacity budget of the coherence controller should be used to keep block sharers in the sparse directory and an indication of the presence of a tag in any private cache in the filter. If we assume a structure similar to fig. 1, (i.e. assigning an independent filter to the private blocks that map onto the same directory entry), the storage cost of the filter and the directory is similar. Additionally, if we do not share the counters between different hashes, i.e. a parallel bloom filter, the number of read/write ports is independent of the number of hash functions. According to [34], the accuracy of this approach is close to a costlier true bloom filter.

For a conventional Counting Bloom Filter (CBF) with n private blocks mapped onto the entry, and m counters with the optimal number of hashes, according to [8], the probability of a false positive is approximately:

$$P_{fp\_CBF} \approx \left(ln(2)\right)^{m/n}$$

Then, to achieve a false positive probability of 5%, we require that $m/n>8$, which is equivalent to having at least 8 counters per private cache entry. To minimize the saturation probability, we need at least 4 bits per counter. In this case, the overflow probability is approximately $e^{-ln2}(ln\ 2)^{16}/16!)=6.8E-17$ [8]. Therefore, we need at least 32 bits per private block. This represents a saving of 50% for a 64-bit tag.

Unfortunately, conventional hash functions behave far from ideally, which unbalances counter usage in a CBF [31]. A quasi-perfect alternative is *d-left hashing* [37]. Over this base, Bonomi et al. [7] introduce a new filter called a *d-left* Counting Bloom Filter (dlCBF) that at least doubles the efficiency of the counters of a CBF, with a similar implementation cost. A high-level representation of this filter is shown in fig. 2. The filter works as follows: the table is divided into as many sub-tables as necessary (two in the example, four in practice might suffice). Each table is divided into "buckets", five in the example. Each bucket is divided into multiple cells. Each cell has a small counter and a few bits to store the address signature. The address signature (called remainder) and the bucket are computed applying a conventional hash function to the address and a permutation per

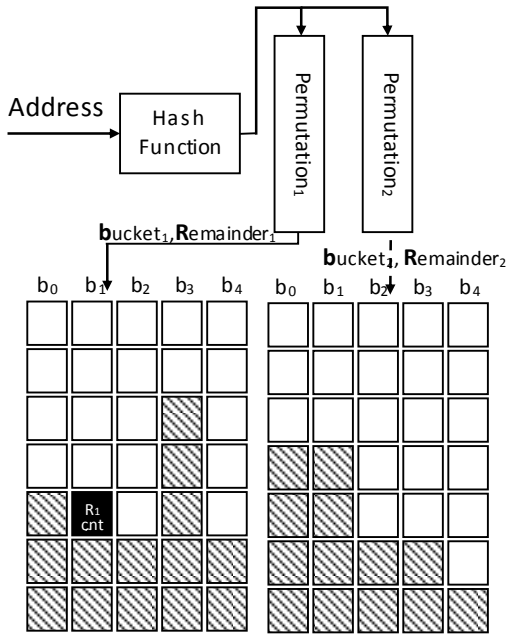

Fig. 2. Sketch of a dlCBF filter. Each cell stores a remainder and a counter.

sub-table to the previous result. Both can be done using simple combinatory logic [34]. The counter is only employed for the unusual case of different addresses with the same remainder (in practice 2 or 3 bits suffice).

When we compute the bucket for each sub-table, we allocate one entry in the least used bucket. The key element in this filter is *d-left* hashing, which establishes that if two destination buckets for an address have the same number of entries, the left one should always be chosen [37]. This allows near-perfect cell load usage, and consequently better coverage for the filter. In the case of the example, the bucket chosen is b_1 for the first sub-table and b_2 for the second one. If we assume that both have the same number of entries allocated (two different remainders with the same number of occurrences), we choose the left table to add the new remainder. If we assume that the new remainder is not present in the table, we allocate a new entry for it and set the cell counter to 1. To look up the filter, the operation is similar. Note that the total number of bits per bucket is pretty small, so the search can be done in an external register avoiding the need for a content addressable memory. Since all sub-tables are handled independently, like in a parallel bloom filter, we can use a conventional single-ported SRAM memory. Therefore, like in a conventional CBF, we can use the same storage to keep either directory or filter information. The controller logic should route each access in accordance with its use. Note that filter updates are done on LLC replacement or on-chip misses; in the first case, outside the critical path, and in the second one, overlapped with main memory access. Since the filters are banked, even in the case of having thousands of pending on-chip memory operations, the number of pending operations per filter will be low. Consequently, we can assume that the cost of updating the filter will be negligible. The look-up of a filter with A sub-tables is similar to a conventional parallel counting bloom filter with A hash functions: each bucket in the corresponding sub-table is read in parallel. We should look up the presence of the remainder in the bucket. This is equivalent to a tag search in an A-associative cache, we assume that the look-up operation is the same as that required for an equivalent directory.

For a dlCBF, the probability of false positives, with a d sub-table configuration of b buckets each, and r bits per remainder is given by [7]:

$$P_{fp\_dlCBF} = 1 - (1 - \frac{1}{d \cdot b \cdot 2^{-r}})^n \approx \frac{n}{d \cdot b} 2^{-r}$$

Assuming c cells per bucket and a utilization factor of ρ (<1), and choosing a total number of buckets depending on n as $n/(\rho \cdot c \cdot d)$:

$$P_{fp\_dlCBF} \approx \rho \cdot c \cdot d \cdot 2^{-r}$$

Thus, assuming four sub-tables (d=4), eight cells per bucket (c=8) and a 3/4 usage per bucket (ρ=0.75), to achieve a 5% false positive probability, the number of bits per remainder is 9. Assuming a 3-bit counter, each cell requires 12 bits. To compensate for bucket utilization, we have to multiply this number by 1/ρ ~16 bits which is necessary per element tracked. This, is less than half the number required by the conventional bloom filter mentioned above.

In practice, for a 64-bit physical tag (which might include sharing vector and block state), this will lead to a 77% saving. For a 256KB L2 and 32 KB L1I, 32 KB L1D and block sizes of 64 bytes, we will require ~9.2KB per core (less than 3% of the tracked cache size), whereas a conventional CBF will require about 20KB. For this configuration each bucket uses 128 bits, which simplifies table lookup. If we reduce the filter size (to save area), the false positive probability might increase, but not in a significant way. Note that all the theoretical estimations are made assuming no spatial locality or sharing in the stream of addresses.

In regard to the overflow probability of the 2-bit counter, a higher bound for the configuration assumed with 5k blocks per private cache, with *cnt=3* bits per counter, is given by [7]:

$$\binom{n}{2^{cnt}+1}\left(\frac{1}{\#buckets \cdot 2^r}\right)^{2^{cnt}+1} \approx \binom{5000}{9}\left(\frac{1}{256 \cdot 2^9}\right)^9 \approx 4.6 \cdot 10^{-19}$$

In any case, we have provided a fallback mechanism to rescue the system in this situation. Note that the system could be running continuously for months or years, and so the inclusion of this corner case is not an option.

B. Filter/Directory Resource Partitioning

The filter and the sparse directory are "complementary" structures, since actively shared blocks will reduce the load on the filter and private blocks will never use directory entries. Initially, we will assume storage resources statically splitting the available SRAM for one or other purpose. Restricting the filter capacity will increase the probability of false positives, which will increase the on-chip traffic and delay off-chip misses. Restricting the directory capacity to the point that the working set of the actively shared blocks does not fit will increase the reconstruction probability. This could be used effectively to provide a low tracking capability (perhaps for less than 20% of private cache blocks).

dlCBF is able to reconstruct the members from the remainder and bucket information. To do so, the permutation used to fill up the tables has to be invertible [7]. Then, it is possible to reconstruct a sub-table by observing the remainder sub-table and simply rolling back the permutation. Thus, it is possible to "move" the tracking of a block from one sub-table to another. In this way, if we detect that the directory is highly loaded (through the frequency of reconstructions), we can adaptively de-allocate one of the entries in the filter and expand the number of ways in the directory. Similarly, if the workload is not using the directory entries because all/most of the data are private, we can expand the filter with additional sub-tables, reducing the false positives without increasing the reconstruction frequency. Note that this operation is not possible in a conventional CBF.

Initially, for all workloads, we will equally split the storage capacity between the filter and the directory. Later, and after observing the application's behavior, we will explore workload-dependent partitioning in order to see how beneficial this approach can be in reducing adverse effects in extreme cases. We will not discuss implementation cost, although it does not seem to be an issue. Perhaps some multiplexors will be necessary to connect the SRAM content to the coherence controller or the filter. In any case, note that the dynamic

behavior will have a negligible impact on performance since the one-by-one migration of filter sub-table entries to directory could be done in the background with the normal system operation. The opposite operation is direct (just requires invalidating directory entries that will be used as a sub-table). We will not discuss the mechanism for triggering this process since the sharing pattern of the applications is quite stable throughout the execution. The most cost effective way is to trigger it by software at the beginning of the workload.

V. EVALUATION METHODOLOGY

A. System Configuration

To analyze FLASK, we model a CMP with out-of-order cores that mimics the execution resources and on-chip cache hierarchy of the Intel Haswell processor [16]: using 6-wide issue cores with 196 in-flight instructions and up to 64 pending memory operations. The number of cores in the CMP is 16. Therefore, the coherence fabric has to support up to 1024 concurrent memory operations. There are three levels of cache. The first two are private, strictly non-inclusive (i.e. L2 acts as a victim cache of L1). The third level is shared and uses a mesh network, which is characterized by better on-chip bandwidth scalability than a ring network. We will assume that the routers in the network can handle multicast traffic natively [19], have single-cycle low-load pass-through [23], separate virtual-networks to avoid end-to-end protocol deadlock and over-provisioned buffering (90 flits per port). Similarly to the LLC, the directory is banked and interleaved by the least significant bits. To quantify FLASK's properties and to understand how it behaves compared to snoop and directory-based protocols, we have implemented two reference protocols based on TokenB [24] and on a sparse-directory [15] respectively. TokenB has been selected because it allows a scalable out-of-order network to be used without adding additional mechanisms outside the coherence controllers. Using the same methodology and tools, all coherence protocols have been optimized fairly. A full SLICC specification for a 3-level hierarchy can be found in [38]. Memory bandwidth is over-provisioned to avoid the necessity of tuning memory controller architecture to each protocol. In practice, contention is negligible in all protocols. In a more realistic environment, broadcast protocol should be handled carefully to avoid unnecessary memory requests wasting a very scarce resource such as off-chip bandwidth [10].A summary of the other main parameters used in our analysis is shown in Table I.

B. Workloads & Simulation Stack

We will use GEMS [26] as the main tool for our evaluation. With GEMS, it is possible to perform full-system simulations. Coherence protocols have been implemented using the SLICC language (Specification Language for Implementing Cache Coherence). In order to model accurately interconnection network contention and its impact on the average access time, we replace the original network with TOPAZ [1]. For power and cost modeling, we use CACTI 6.5 [29] for the cache and DSENT [36] for the network. Ten workloads, shown in Table II, are considered in this study, including both multi-programmed and multi-threaded applications (scientific and server) running on top of the Solaris 10 OS. The numerical applications are three of the NAS Parallel Benchmarks suite (*OpenMP* implementation version 3.4 [20]). The server benchmarks correspond to the whole Wisconsin Commercial Workload suite [2]. The remaining class corresponds to multi-programmed workloads using part of the SPEC CPU2006 suite [35] running in rate mode (where one core is reserved to run OS services). The mix of workloads has been selected trying to cover diverse usage scenarios, varying the sharing degree (from none in SPEC applications to a large amount in Server Workloads) and sharing contention (from none in SPEC to a large amount in scientific applications). Among the NAS applications, we chose the three with the highest sharing contention. From the SPEC suite, we chose three applications with a variable range in working set size. We should emphasize that the three families of applications exhibit quite dissimilar behavior from the coherence protocol perspective, but they have to be considered, given the usage scenarios of general purpose CMPs. Focusing the evaluation on a single suite of benchmarks, it is hard to consider all the characteristics we have represented with the selected mix.

We model hardware-assisted TLB fill and register window exceptions for all target machines. Multiple runs are used to fulfill strict 95% confidence intervals (error bars are not visible in most cases). Benchmarks are fast-forwarded to the point of interest, during which page tables, TLBs, predictors, and caches are warmed up. In iteration-based applications, such as NPB, a

TABLE I. SUMMARY OF 16-CORE CMP SYSTEM CONFIGURATION

Core Arch.	Functional Units	4xI-ALU/4xFP-ALU/ 4xD-MEM
	ROB size / Issue Width	196, 6-way
	Frequency, count	3Ghz, 16 cores
Private Caches	(L1)Size/Associativity / Block Size / Access Time/Repl.	32KB I/D, 4-way, 64B, 1 cycle, LRU
	(L2)Size / Associativity/ Block Size / Access Time/Repl.	256KB Unified, 8-way, 64B, 2 cycles, LRU, Exclusive with L1
	Outstanding Requests per core	64
Shared L3	Size / Associativity / Block Size/Repl.	16MB (or 32MB), 16 (or 32)×1MB, 16-way, 64B , LRU
	NUCA Mapping	Static, interleaved by LSB
	Slice Access Time	6 cycles
Mem	Capacity / Access Time / Memory Controllers / BW	4GB, 240 cycles, 4/32GBs
Network	Topology / Link Latency / Link Width/Clock	4×4 Mesh, 1 cycle, 16B, 3Ghz
	Router Latency (low-load) / Flow Control / Routing/ Buffering	1-3 cycle/ Wormhole/DOR/10KB

TABLE II. MULTITHREADED WORKLOADS

SERVER [2]	OLTP	IBM DB2 DBMS, TPC-C like 10000 Transactions
	Apache	Apache web server, SpecWeb like, 25000 Transactions
	JBB	SpecJBB, 70000 Transactions
	Zeus	Zeus web server, SpecWeb like, 25000 Transactions
NPB[20]	Multi-Grid (MG)	CLASS A
	Fast Fourier Transform (FT)	CLASS W
	LU Diagonalization (LU)	CLASS A
SPEC [35]	Astar	Native, 15 thr.
	Hmmer	Native, 15 thr.
	Omnetpp	Native, 15 thr.

warm checkpoint is taken in the middle of the execution and with a reduced number of iteration runs. Transactional workloads are warmed up by running hundreds of thousands of transactions, and accurately simulated for a fixed number of transactions. SPEC workloads are fast-forwarded to the point of interest and simulate ~8billion instructions.

VI. Performance Results

A. Comparative results with reference protocols

The fundamental parameters of the system for the considered coherence protocols running the selected workloads, namely *execution time*, *average access-time* and memory hierarchy *energy delay product* (EDP) are shown from fig. 3 to fig. 5. To clarify the total storage, we denote it as SDE (*Sparse-Dir Equivalent*) capacity. All the results are normalized against token coherence and the directory size is being swept from a capacity to track 160% of the private cache blocks (8K SDE entries) to just 5% (32 SDE entries). Note that in order to keep implementation cost constant, in FLASK, at this point, half of this capacity is devoted to the filter and half to the directory. In other words, in the most extreme case, the FLASK directory will only have SDE capacity to track 16 shared blocks per controller (256 in the whole chip).

As expected, when the size of the directory is below one fourth of the aggregate private caches' capacity, the sparse-directory performance is degraded. In contrast to other previous works, [12], we observe this degradation with significant directory size reduction. This is related to the fact that in that work there is only one level of private caches. Here, L2 acts as a victim cache for L1. Thus, a directory induced private miss is eight times more frequent in L2 than L1. If we take into account that in L2, block reuse is low [18], the results seem reasonable. With an inclusive L1/L2 (which for an aggressive out-of-order core and a shared LLC might not be interesting), the effects will be more noticeable but still not too acute. Additionally, it should be noted that for an in-order core, directory-induced private misses (and subsequent hits in L3) will have more effect on performance. The memory level of parallelism present in the evaluated system allows the impact to be partially hidden.

As can be seen in fig. 5, sparse directory hits in private caches are reduced when the count of tracked blocks is decreased. Thus, there is a substantial increment as data must be

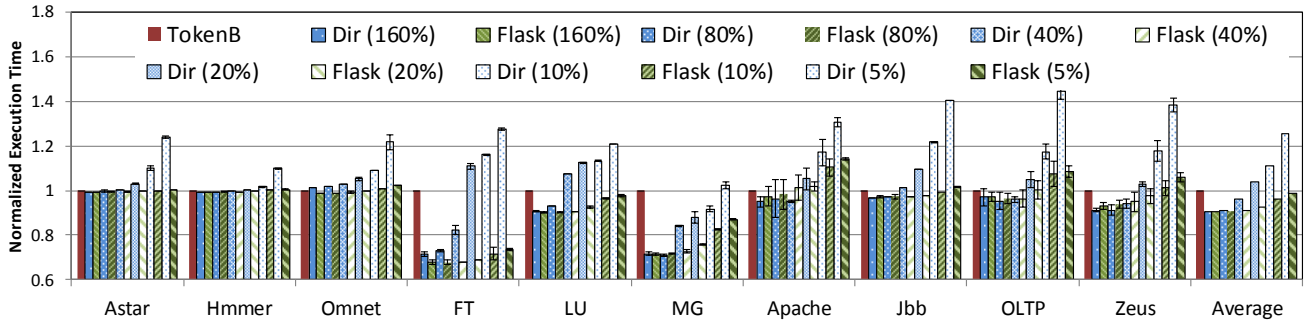

Fig. 3. TokenB Normalized Execution Time.

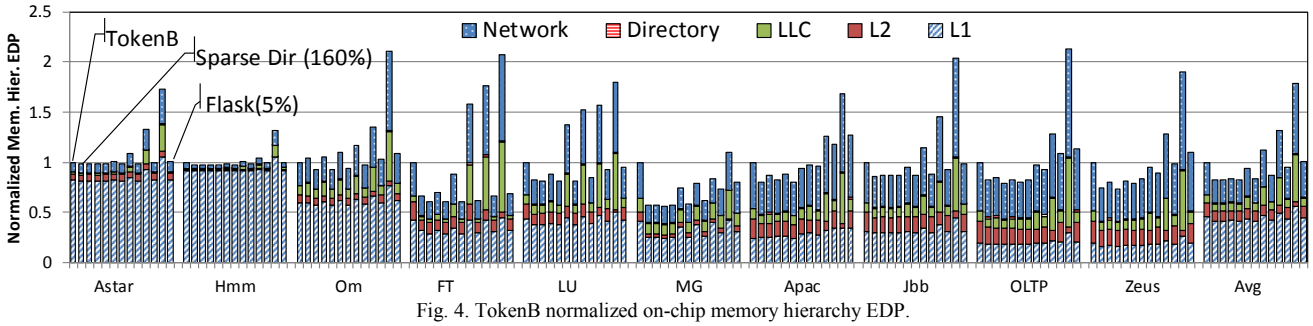

Fig. 4. TokenB normalized on-chip memory hierarchy EDP.

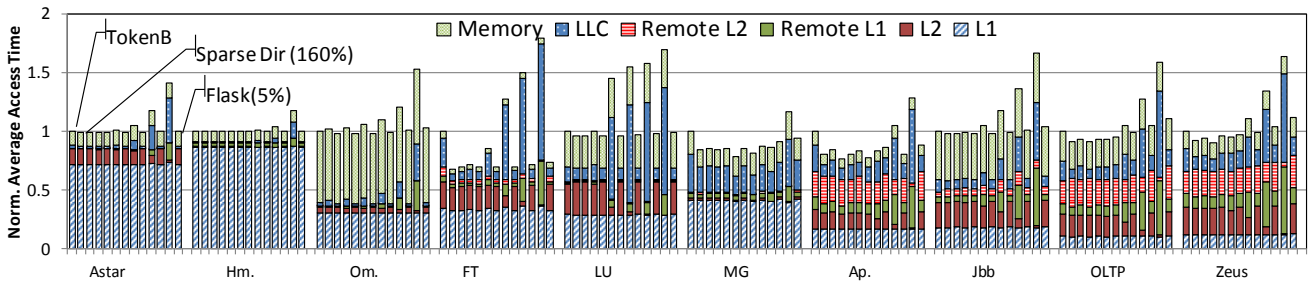

Fig. 5. TokenB normalized average memory access time.

retrieved from LLC (for private blocks) or other private caches (for actively shared blocks). In numerical or multi-programmed workloads the former case is more frequent while the latter exists in server workloads. This is consistent with the sharing degree. As can be seen for applications with a large portion of shared data, the latency degradation from 160% to 5% is almost doubled, degrading the performance by more than 40% on average. For these cases the intense coherence traffic due to directory-induced invalidations and subsequent LLC hits makes the activity increase in the network substantially, which degrades the energy properties, making it worse than the snoop-based protocol.

When we look at the performance of the snoop-based coherence protocol, we can observe unstable behavior. In some applications it seems to be almost the best performer while in others it is clearly not. The reason can be found in the on-chip contention. For some of the scientific applications, there is a significant traffic pressure on the LLC network. In spite of having a reasonably dimensioned network (4x4 mesh with 3 GHz clock and 16Byte links) and a router with state-of-the-art features [19] [23], the latency of the LLC is larger. Despite being an average sized system, extra traffic imposed by broadcast requests and the concurrent memory operations (at a given time there may be more than 16K packets in-flight) seem to surpass network capabilities. The solution for this unpredictable behavior is to oversize the network (e.g. increasing the link width, using topologies with better connectivity, using a more advanced router, etc.) or to redesign the coherence protocol [30]. Unfortunately this would be inappropriate for bigger systems. In any case, even in applications where TokenB has a slight performance advantage, the energy consumption is higher.

FLASK exhibits very different behavior to the other protocols. Even in the most extreme configuration, the performance seems quite unaffected. In the worst case, which is *apache*, the performance degradation observed versus token is 13%. In general, it seems that applications with low sharing degree are correctly handled, with almost negligible performance degradation. Having only SDE capacity to track 256 private blocks seems to affect other applications more significantly. This causes more reconstructions, which delays access to shared data after a capacity miss in private caches, slightly lengthening LLC access time. In other applications, such as the multi-programmed workloads, in spite of having a minimal filter (just 32 buckets per sub-table), the effects both on performance and energy of this extreme configuration are negligible. On average the performance degradation when reducing from 160% SDE capacity to 5% is only 8%, which is a noteworthy result. In all cases, the number of private-cache external invalidations due to the replacement of blocks in LLC without all the tokens is negligible.

Since the on-chip traffic is filtered out, there is no similar behavior to that observed for TokenB. In general, the energy requirements of the protocol are smaller than token and are quite steady regardless of the directory size. Note that this metric is pessimistic, since we exclude the cores' power consumption. As FLASK is the best performer in most cases, the EDP of these components will be low.

In summary, with almost no tracking capability, FLASK is more stable and energy efficient than broadcast-based protocol, and its performance degrades much more gracefully than conventional sparse-directory when directory size is reduced.

B. *Filter Efficiency*

The previous results contain some details that we want to highlight, namely, the filter efficiency. The main figure of merit is the false positives, which increase on-chip energy and memory

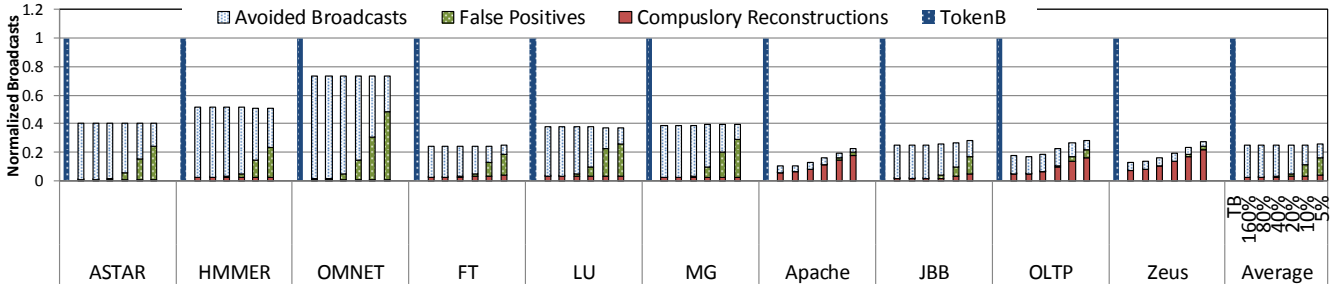

Fig. 6. TokenB normalized filter efficiency for different SDE sizes.

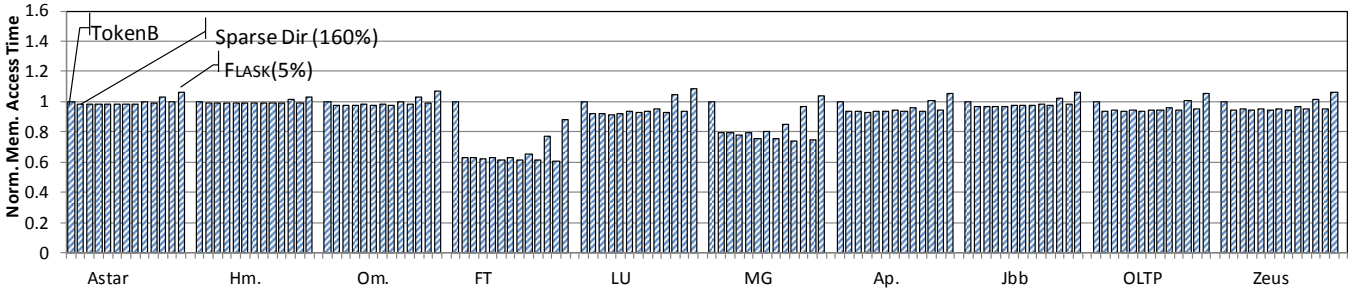

Fig. 7. TokenB normalized memory latency overhead.

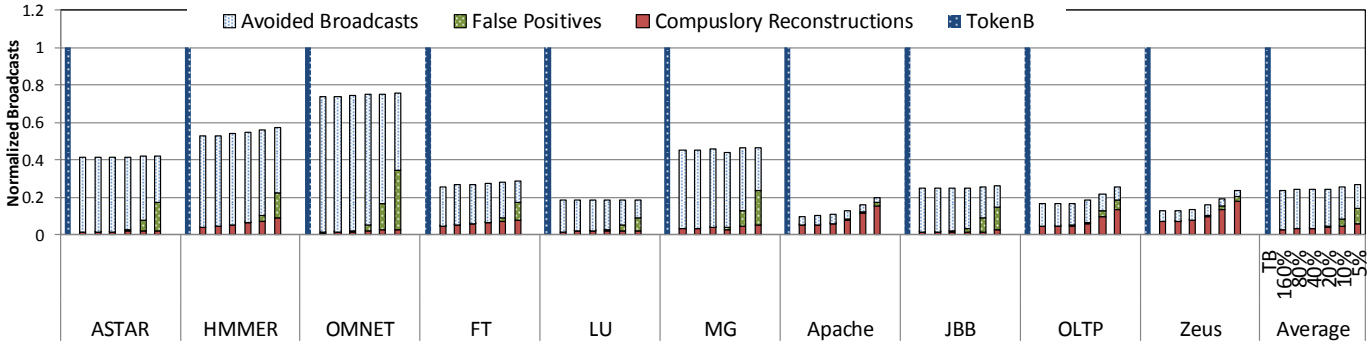

Fig. 8. TokenB normalized traffic filtering with adaptive partitioning.

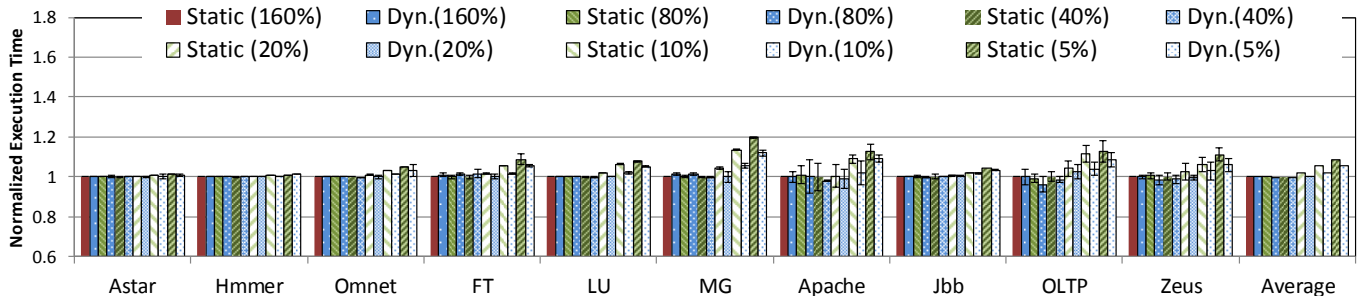

Fig. 9. 40% SDE Static normalized performance of FLASK with dynamic and static storage allocation.

access time. Fig. 6 shows the total number of multicasts over a range of available storage for the filter capacity.

For reference, we normalize this metric to multicast in TokenB. Note that, when we take into account that the network can natively handle multicast traffic and the cache snoop energy contribution, this might not be directly transferred to link utilization or energy consumption, as we can contrast these results with fig. 4. When the directory is dimensioned for 20% of private caches, false positives seem to be consistent with the theoretically expected proportion of 5%. If we shrink the SDE capacity to just 5% of private caches, although there is an increase in false positives, counter saturation never occurs in any of the runs of our evaluations. While in some cases, even with such a small filter, it is able to detect a reasonable number of on-chip misses; in other cases (such as the NAS applications), it cannot do so. In some applications, the number of true positives (private cache misses for actively shared data) grows when we reduce the size of the directory while in others, it remains almost unchanged. In the former, the directory is not able to maintain the shared working set, whereas in others it can do so. In contrast to this behavior, false positives are more numerous in the latter applications as private blocks increase the number of elements to be tracked by the filter.

Fig. 7 shows the TokenB normalized memory access time (from the core perspective, i.e. includes the whole latency to transmit the desired word from memory to the processor back-end) for FLASK and sparse-dir. Note that memory requests in both cases will be roughly the same, since most off-chip requests are induced by LLC capacity misses and both protocols handle LLC data in the same way. In numerical applications, false positives for very small filter capacities have a negative effect due to the delay in off-chip access and the on-chip latency that the extra traffic adds. Consequently, there is a significant increase in memory access time. In applications such as the server workloads, the extra traffic due to compulsory reconstructions increases the contention, increasing the access time to the memory controller and therefore delaying the memory access.

For the generous off-chip bandwidth considered, the results are dominated by on-chip effects: on-chip contention and the additional latency induced by false positives (which delays the memory request until the coherence controller realizes that there is no on-chip copy). Even in the most adverse directory configuration, the effect is less than 5%. This is almost unnoticeable in the average access time, as can be appreciated in fig. 5. Note that in a system with many cores or large private caches, this configuration could reduce the storage footprint significantly. In any case, directory size can be a useful knob for the trade-off between implementation and energy cost.

C. *Adaptive Filter/Directory resource partitioning*

As stated before, dlCBF allows adaptive resource splitting. Although we did not implement the on-line adaptation, we can statically choose the best configuration for the application. Looking at fig. 6, it seems clear that multi-programmed, numerical applications require few entries in the directory. On the contrary, for commercial workloads, the on-chip traffic is dominated by reconstructions. Therefore, we should select only a 1-way directory for the first two classes (with 7 sub-tables in the filter) and a 7-way directory for the last class (with just 1 sub-table in the filter). Note that even in a multi-programmed workload, a small number of OS addresses might be shared, so we need at least a 1-way directory. The broadcast signature, normalized against the token, is shown in fig. 8. The

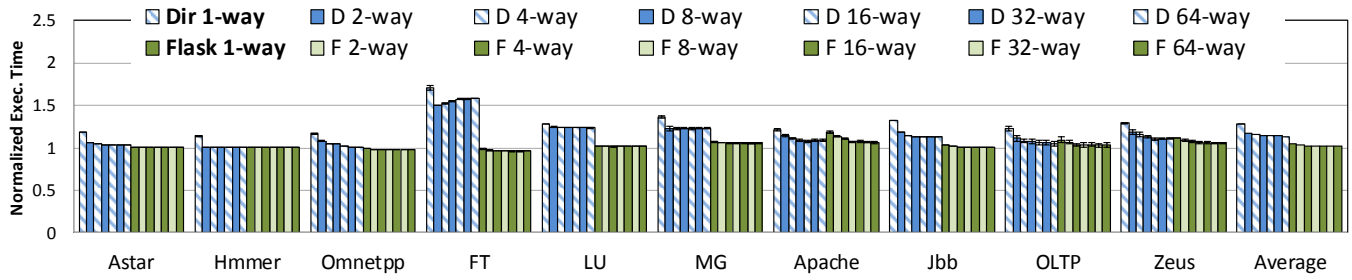

Fig. 10. 160% SDE sparse directory normalized performance for different associativities (20% SDE).

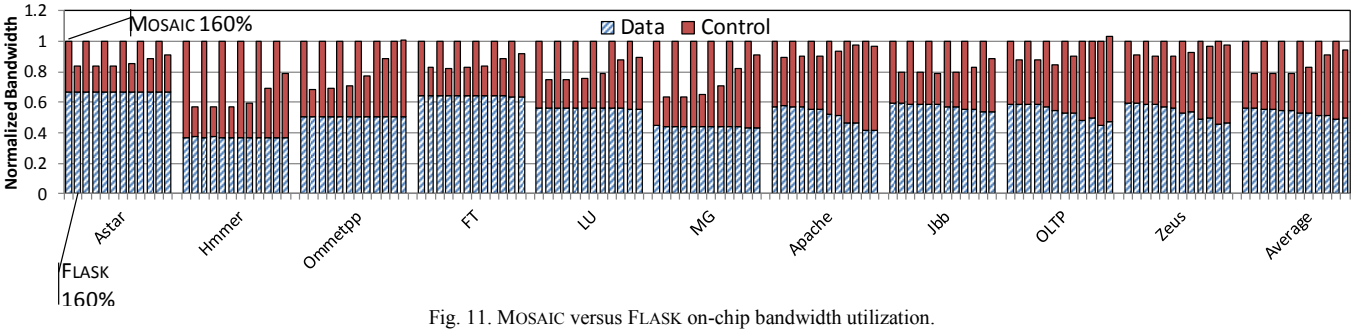

Fig. 11. Mosaic versus Flask on-chip bandwidth utilization.

performance results of the static and dynamic storage allocation in the memory controller are shown in fig. 9. If we compare these results with those provided in fig. 6, the effect of the approach is to halve the storage resources with negligible impact. For example, under these conditions with 10% SDE capacity, the traffic requirements are close to 20% of SDE capacity when the results are statically halved. Consequently with just 10%, in most applications, a tolerable false positive rate is observed. On average, with 10% and 5% of SDE capacity, we observe a performance penalty of about 3% and 6% respectively. Although dlCBF is suitable for an on-line mechanism, capable of morphing sparse-directory and filter throughout the application execution, we left this analysis open for future work. Under cloud-computing scenarios (with live workload migration), this might be interesting, given the results seen here. However, in any case, we believe that such a task should be done in the software layer, since the switch in behavior will be quite infrequent.

D. Comparing Flask with other directory cost reduction alternatives

Flask can be combined with other strategies focused on the same goal [11][12][13][27][32]. Some of them are focused on eliminating the directory overprovision by emulating large associativity through multi-hashing indexing and insertion [32] [33], achieving in most workloads the benefits of a very high associativity at a fraction of the cost. Therefore, if we increase the associativity of the sparse directory, we can achieve a similar effect. Fig. 10 shows the over- provisioned normalized result for different associativities (ranging from 1-way to 64-way with a fixed capacity). In order to appreciate conflict misses in the directory, and given that the computational requirements of the evaluation framework system and application scalability prohibit it, we artificially reduce the size to just 20% SDE (otherwise evictions caused by conflict in the directory are negligible). Except in the case of FT, whose counterintuitive behavior is caused by very low directory reuse [32], increasing the associativity reduces the directory conflicts, which provides a slight performance degradation. As we can appreciate, 1-way directory in Flask is able to outperform the sparse directory even with 64 ways. In Flask, the impact of directory conflicts on performance is negligible. Therefore, Flask will provide better performance than techniques focused on minimizing directory conflicts such as [11][12]. The reasons for this are: (1) there is no need to perform external invalidations after a directory eviction, and (2) it only uses the directory to track actively shared blocks. Under such circumstances, the experimental observations made by [15] about conflicts are no longer applicable. In fact, although not exploited here, Flask can be used to reduce directory implementation cost (v.gr. using very low associativity).

Like Flask, Mosaic was focused on improving directory scalability, eluding directory inclusiveness through entry reconstruction on demand. Fig. 11 compares the on-chip memory bandwidth consumption for the two approaches, for different SDE capacities. As expected, the control traffic generated by Flask is significantly less than Mosaic. This is because directory entry reconstructions in Flask are performed only for shared data. This means that in some multi-programmed workloads, such as *hmmer,* the total amount of traffic is up to 60% less. When the size of the filter is reduced, the false positives increase the traffic slightly. With applications with a high sharing degree, such as *apache,* this advantage is smaller. In this last type of benchmarks, when the size of directory is reduced below 20%, the behavior of the two protocols becomes more similar due to the fact that most reconstructions are because of shared blocks. On average, and with a directory of at least 20%, Flask enables the reduction of total on-chip traffic by 20%.

VII. CONCLUSIONS

We have proposed an evolutionary re-architecture for directory coherence protocols that can benefit from snoop-based coherence without paying a high toll. The results enable a balanced approach that improves system performance in a wide range of applications.

The proposal might be beneficial to scale the coherence protocol for many-core systems or for medium-size CMPs, as we have demonstrated in the results section especially so bearing in mind recent and forthcoming commercial systems. In any case, we have shown that even for 16-core CMPs there are both power and performance benefits versus other protocols.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their helpful comments. Special thanks to Dan Sorin and Viji Srinivasan for their valuable suggestions.

REFERENCES

[1] P. Abad et al., "TOPAZ: An Open-Source Interconnection Network Simulator for Chip Multiprocessors and Supercomputers," in *Int. Symposium on Networks-on-Chip (NOCS)*, 2012, pp. 99–106.

[2] A. R. Alameldeen et al., "Simulating a $2M commercial server on a $2K PC," *Computer (Long. Beach. Calif).*, vol. 36, no. 2, pp. 50–57, Feb. 2003.

[3] M. Alisafaee, "Spatiotemporal Coherence Tracking," in *Int. Symposium on Microarchitecture (MICRO)*, 2012, pp. 341–350.

[4] J. Almeida and A. Z. Broder, "Summary cache: a scalable wide-area Web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[5] I. Atta, P. Tözün, X. Tong, A. Ailamaki, and A. Moshovos, "STREX," in *Int. Symp. on Computer Architecture (ISCA)*, 2013, vol. 41, no. 3, p. 273.

[6] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[7] F. Bonomi, M. Mitzenmacher, and R. Panigrahy, "An improved construction for counting bloom filters," in *ESA'06 14th Annual European Symposium*, 2006, pp. 684–695.

[8] A. Broder and M. Mitzenmacher, "Network Applications of Bloom Filters: A Survey," *Internet Math.*, vol. 1, no. 4, pp. 485–509, Jan. 2004.

[9] M. Butler, "AMD 'Bulldozer' Core - a new approach to multithreaded compute performance for maximum efficiency and throughput," in *Symposium on High-Performance Chips (HotChips)*, 2010.

[10] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes, "Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor," *IEEE Micro*, vol. 30, no. 2, pp. 16–29, Mar. 2010.

[11] B. A. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. F. Duato, "Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks," in *Int. Symp. on Computer Architecture (ISCA)*, 2011, p. 93.

[12] S. Demetriades and S. Cho, "Stash Directory: A Scalable Directory for Many-Core Coherence," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[13] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo directory: A scalable directory for many-core systems," in *2011 IEEE 17th Int. Symp. on High Performance Computer Architecture*, 2011, pp. 169–180.

[14] E. J. Fluhr et al., "POWER8: A 12-core server-class processor in 22nm SOI with 7.6Tb/s off-chip bandwidth," in *International Solid-State Circuits Conference (ISSCC)*, 2014, pp. 96–97.

[15] A. Gupta, W. Weber, and T. Mowry, "Reducing memory and traffic requirements for scalable directory-based cache coherence schemes," in *Int. Conference on Parallel Processing (ICPP)*, 1990, pp. 312–321.

[16] P. Hammarlund et al., "Haswell: The Fourth-Generation Intel Core Processor," *IEEE Micro*, vol. 34, no. 2, pp. 6–20, 2014.

[17] J. Huh et al., "A NUCA substrate for flexible CMP cache sharing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 8, pp. 1028–1040, 2007.

[18] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Int. Symposium on Computer Architecture (ISCA)*, 2010, pp. 60–72.

[19] N. E. Jerger, L. S. Peh, and M. Lipasti, "Virtual circuit tree multicasting: A case for on-chip hardware multicast support," in *International Symposium on Computer Architecture (ISCA)*, 2008, pp. 229–240.

[20] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance," *NASA Ames Research Center," Technical Report NAS-99-011*, Citeseer, 1999.

[21] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE Micro*, vol. 25, no. 2, pp. 21–29, Mar. 2005.

[22] T. Krishna, L.-S. Peh, B. M. Beckmann, and S. K. Reinhardt, "Towards the ideal on-chip fabric for 1-to-many and many-to-1 communication," in *Int. Symposium on Microarchitecture (MICRO)*, 2011, vol. 2, pp. 71–80.

[23] A. Kumary, P. Kunduz, A. P. Singhx, L.-S. Pehy, and N. K. Jhay, "A 4.6Tbits/s 3.6GHz single-cycle NoC router with a novel switch allocator in 65nm CMOS," in *International Conference on Computer Design (CCD)*, 2007, pp. 63–70.

[24] M. M. K. Martin, M. D. Hill, and D. A. Wood, "Token Coherence: decoupling performance and correctness," in *30th Annual International Symposium on Computer Architecture, 2003. Proceedings.*, 2003, pp. 182–193.

[25] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood, "Bandwidth adaptive snooping," in *International Symposium on High Performance Computer Architecture (HPCA)*, pp. 251–262.

[26] M. M. K. Martin et al., "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Comput. Archit. News*, vol. 33, no. 4, p. 92, Nov. 2005.

[27] L. G. Menezo, V. Puente, and J. A. Gregorio, "The case for a scalable coherence protocol for complex on-chip cache hierarchies in many-core systems," in *Int. Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013, pp. 279–288.

[28] A. Moshovos, "RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence," in *Int. Symp. on Computer Architecture (ISCA)*, 2005, pp. 234–245.

[29] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *Int. Symposium on Microarchitecture (MICRO)*, 2007, pp. 3–14.

[30] A. Raghavan, C. Blundell, and M. M. K. Martin, "Token tenure: PATCHing token counting using directory-based cache coherence," in *Int. Symposium on Microarchitecture (MICRO)*, 2008, pp. 47–58.

[31] M. V. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *IEEE Trans. Comput.*, vol. 46, no. 12, pp. 1378–1381, 1997.

[32] D. Sanchez and C. Kozyrakis, "SCD: A scalable coherence directory with flexible sharer set encoding," in *18th IEEE Int. Symposium on High Performance Computer Architecture*, 2012, pp. 1–12.

[33] D. Sanchez and C. Kozyrakis, "The ZCache: Decoupling Ways and Associativity," in *MIcro 2010*, 2010, pp. 187–198.

[34] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam, "Implementing Signatures for Transactional Memory," *Int. Symp. Microarchitecture*, pp. 123–133, 2007.

[35] SPEC Standard Performance Evaluation Corporation, "SPEC 2006." .

[36] C. Sun et al., "DSENT - A Tool Connecting Emerging Photonics with Electronics for Opto-Electronic Networks-on-Chip Modeling," in *Int. Symposium on Networks-on-Chip (NOCS)*, 2012, pp. 201–210.

[37] B. Vöcking, "How asymmetry helps load balancing," *J. ACM*, vol. 50, no. 4, pp. 568–589, Jul. 2003.

[38] "SLICC specification of Flask and Counterpart Coherence Protocols." [Online]. Available: http://www.atc.unican.es/galerna/flask.

Transactionalizing Legacy Code: an Experience Report Using GCC and Memcached *

Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear
Lehigh University
{wer210, trv211, yul510, spear}@cse.lehigh.edu

Abstract

The addition of transactional memory (TM) support to existing languages provides the opportunity to create new software from scratch using transactions, and also to simplify or extend legacy code by replacing existing synchronization with language-level transactions. In this paper, we describe our experiences transactionalizing the memcached application through the use of the GCC implementation of the Draft C++ TM Specification. We present experiences and recommendations that we hope will guide the effort to integrate TM into languages, and that may also contribute to the growing collective knowledge about how programmers can begin to exploit TM in existing production-quality software.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

Keywords Transactional Memory; memcached; GCC; C++

1. Introduction

For over a decade, Transactional Memory (TM) [13, 31] has been promoted as an efficient programming idiom for simplifying the creation of concurrent software. In recent years, the C/C++ community has made great strides toward integrating TM into mainstream products, and today there is both a draft specification [1] and a set of reference compiler implementations (GCC, Intel CC, LLVM, and xlC).

The integration of TM into C++ enables its use in two directions. First, it allows programmers to create new software from scratch that is designed around transactional constructs. Second, it enables existing software to be retrofitted with transactions, either to simplify the creation of new features, or to improve (from a performance or maintenance perspective) existing code. In the former category, studies by Rossbach et al. [28] and Pankratius and Adl-Tabatabai [24] have shown that TM can simplify the creation of new software. Similarly, there are several microbenchmark and benchmark suites that demonstrate the use of TM, most notably STAMP [23], EigenBench [15], Atomic Quake [37], Lee-TM [2], SynQuake [20], and RMS-TM [16].

While these benchmarks and studies collectively provide a good platform for evaluating TM implementations, they treat the STM interface as a constant. Our effort in this paper is complementary: we are interested in evaluating the proposed C++ TM interface. Our approach is to transactionalize a real-world application using the Draft C++ TM Specification, with an eye towards identifying (a) what common programming patterns and idioms are not well supported, (b) what features are cumbersome or difficult to use, and (c) what performance implications the specification carries.

For the purpose of this study, we used the open-source GCC compiler to replace locks with transactions in the popular memcached in-memory web cache. We used the developmental GCC version 4.9.0, which implements the Draft C++ TM Specification, for two reasons. First, it is widely available and easy to modify, which suggested that if we encountered bugs, we would be able to remedy them quickly. Second, its relatively transparent TM library enabled us to investigate the behavior of different TM algorithms and low-level design decisions.

Similarly, we chose memcached for two reasons. First, it is a popular and realistic application, but one that is still of a tractable size. It was not unreasonable to analyze all language-level locks within the application, or to conduct whole-program reasoning about the correctness of code transformations. Second, it is sufficiently complex to represent a challenge for TM: locks and condition variables are intertwined; there is a statically defined order in which locks must be acquired; there is a reference counting mechanism that employs in-line assembly and volatile variables (i.e.,

* This work was supported in part by the National Science Foundation through grants CNS-1016828 and CCF-1218530.

ASPLOS '14, March 1–5, 2014, Salt Lake City, Utah, USA.

http://dx.doi.org/10.1145/2541940.2541960

C++11 `atomics`); and it uses high-performance external libraries, such as a worklist implemented via libevent [21]. Furthermore, memcached uses some of these primitives in unconventional ways. For example, while there is a strict locking order (item, cache, slab, and then stats locks), this order is sometimes violated by using `trylock` to acquire an item lock while holding the cache lock. Similarly, in some cases a pthread lock is used as a spinlock, by acquiring it using a trylock inside of a while loop.

In our study, we used memcached version 1.4.15, which contains recent scalability enhancements. While our work is similar to a recent effort by Pohlack and Diestelhorst [26], it differs in that we are using memcached to analyze the Draft C++ TM Specification, rather than to assess the utility of a particular hardware TM mechanism.

The remainder of this paper is organized as follows. Section 2 describes the key features of the Draft C++ TM Specification that we evaluate in this paper. Section 3 then chronicles the steps we took to transactionalize memcached, and reports the performance impact of our efforts. In Section 4, we explore changes to GCC that would affect our results. Section 5 presents recommendations for designers and implementors of TM libraries, and Section 6 makes recommendations for programmers intending to use the Draft C++ TM Specification. Section 7 concludes.

2. The Draft C++ TM Specification

The Draft C++ TM Specification [1] integrates transactional memory support into C++ through the addition of several new keywords and annotations. The extensions can be roughly broken down into three categories.

Transaction Declarations: Most substantially, the specification introduces the `__transaction_atomic` and `__transaction_relaxed` keywords. These keywords are used to indicate that the following statement or lexically scoped block of code is intended to execute as a transaction. There are numerous interpretations of the difference between these two keywords. For our purposes, *atomic* transactions can be thought of as being statically checked to ensure that they contain no unsafe operations. While the specification does not guarantee that the meaning of *unsafe* will not evolve over time, a reasonable approximation is to assume that atomic transactions cannot perform I/O, access volatile variables (i.e., C++ `atomics`), or call any function (to include inline assembly) that the compiler cannot prove will be safely rolled back by the TM library if the calling transaction aborts.

Relaxed transactions do not carry the same restrictions as atomic transactions; they are allowed to perform I/O and other unsafe operations. This is achieved by enabling a relaxed transaction to become irrevocable [34, 36], or perhaps to run in isolation, starting at the point where it attempts an instruction that the compiler is not certain can be undone. Such transitions are invisible, though they may present a scalability bottleneck and can introduce the possibility of deadlock (e.g., if two relaxed transactions attempt to communicate with each other through `atomic` variables, then since both must become serial and irrevocable, they cannot run concurrently). However, in the absence of such unsafe code, the two types of transactions are indistinguishable; both should scale equally well.

In addition, the specification supports *transaction expressions*. Transaction expressions are syntactic sugar to simplify code such as using a transaction to initialize a variable, or using a transaction to evaluate a conditional.

Function Annotations: The specification supports two function annotations, `transaction_safe` and `transaction_callable`. A *safe* function is one that contains no *unsafe* operations. That is, it can be called from an atomic transaction. The compiler statically checks that safe functions only call safe functions, and that atomic transactions only call safe functions. In addition, the compiler must generate two versions of any safe function: the first is intended for use outside of transactions; the second contains instrumentation on every load and store, so that the function can be called from within a transaction and safely unwound upon abort. The compiler generates an error if it encounters a function that is marked *safe* but contains unsafe operations.

The *callable* annotation indicates to the compiler that a function will be called from a transactional context, but is not safe. We interpreted this as indicating that it is possible, but not guaranteed, that the function will call unsafe code. This, in turn, means the function can only be called from relaxed transactions. In contrast to the `transaction_safe` annotation, `transaction_callable` is strictly a performance optimization. An implementation is free to execute all relaxed transactions serially, or to try to execute them concurrently as long as no running transaction requires irrevocability in order to perform an unsafe operation. If a relaxed transaction attempts to call a function that the programmer has not annotated as `callable` or `safe`, then unless the compiler has inferred the safety of that function, the transaction must become serial and irrevocable.

Based on this interpretation, we concluded that if a function cannot be marked `safe`, on account of possibly performing I/O or some other unsafe operation, then it should be marked `callable` to ensure that calls to that function from a relaxed transaction do not cause serialization in those instances where the function does not perform an unsafe operation. As an example, consider the following code segment:

```
1  __transaction_relaxed {
2    ...
3    if (verbose)
4      fprintf(stderr, message);
5    ...
6  }
```

When `verbose` is false, this transaction need not become irrevocable. When it is true, the inability of the TM system

to undo writes to `stderr` necessitates that it become irrevocable before calling `fprintf`. Regardless of the value of `verbose`, it must be relaxed: the compiler cannot guarantee that it will never require irrevocability.

Exception Support: The third category of extensions in the Draft C++ TM Specification pertain to exceptions. When a transaction encounters a `throw` statement, failure atomicity may require the transaction to undo all of its effects; in other cases it may be desirable for the transaction to commit its partial state. To express this difference, the specification provides the `__transaction_cancel` statement.

Clearly, an irrevocable relaxed transaction cannot undo its effects, and indeed it is not correct in the current specification for any relaxed transaction to cancel itself. Atomic transactions may explicitly cancel themselves, and may even do so in the absence of exceptions. This, however, creates a challenge: with separate compilation, the compiler may not be able to determine whether a `transaction_safe` function called by a relaxed transaction will attempt to cancel. To remedy the problem, an additional `may_cancel_outer` annotation is required on some safe functions.

Extensions For the purposes of our work, it is useful to consider two extensions to the Draft C++ TM Specification, both of which are supported in GCC. The first is the `transaction_pure` annotation, which allows programmers to indicate that certain functions are transaction safe without their memory accesses being instrumented. While this is intended as a performance optimization, its implementation is not checked: for example, one could annotate `printf` as being `transaction_pure`, and then the compiler would allow calls to `printf` from within a transaction.

The second extension allows registration of functions to run after a transaction commits or aborts. Functions registered as `onCommit` handlers run after a transaction commits. Functions registered via `onAbort` handlers run after an aborted transaction had undone any memory effects, but before it retries. Both take a single untyped parameter.

3. Transactionalizing Memcached

The Draft C++ TM Specification appears to offer a simple route to transactionalizing legacy code: one need only replace all lock-based critical sections with relaxed transactions. Since the existing lock-based code does not require compiler support to undo effects, there is no need for `__transaction_cancel`. Consequently, it would seem that atomic transactions are unnecessary, and `transaction_callable` annotations optional.

There are two flaws in this line of thinking. First, many programs contain condition variables, which require the use of an associated mutex. Dealing with condition synchronization currently requires ad-hoc solutions, and we discuss our solution for memcached below. Secondly, relaxed transactions are prone to serialization when they encounter an unsafe operation. Currently, there are no tools for easily identifying serialization points in relaxed transactions. Thus we claim that programmers should think of relaxed and atomic transactions as differing in terms of the performance model they carry: relaxed transactions have no performance guarantees, but atomic transactions guarantee that serialization will be avoided wherever possible.

Given this performance model, programmers can replace relaxed transactions with atomic transactions to gain a static guarantee that those transactions will not cause unnecessary serialization. To guide this effort, the programmer can use error messages from incorrect uses of atomic transactions and `transaction_safe` attributes as a tool for identifying unsafe operations.

We eliminated all contended locks in memcached, without requiring relaxed transactions in the final code. We first identified the locks that should be replaced, and then modified any condition synchronization built atop those locks. Next, we applied the Draft C++ TM Specification to the fullest extent possible to replace locks with atomic or relaxed transactions. We then developed transaction-safe alternatives to unsafe standard libraries. Finally, we applied `onCommit` handlers to eliminate all remaining relaxed transactions, resulting in a program in which no transaction required serialization to complete.

3.1 Identifying Locks

Our first step was to identify those locks that indeed were worthy of removal. There are four categories of locks in memcached, which are acquired in the following order:

1. `item` locks: These locks protect individual elements in the hash table that serves as the central data structure in the application.
2. `cache_lock`: This lock prevents concurrent modifications to the structure of the hash table (i.e., resizing).
3. `slabs_lock`: This lock protects the slab allocator. Slabs are memory blocks that store sets of objects of the same maximum size.
4. `stats_lock`: This lock protects a set of counters for program-wide statistics. While much effort has gone into moving these counters into per-thread structures, some remain as global variables.

We profiled the contention on locks by using mutrace [25]. This revealed that the `cache_lock` and `stats_lock` were the only locks that threads frequently failed to acquire on their first attempt. Thus it was necessary to replace at least these two locks with transactions. However, several more locks ultimately required replacing:

There were three instances in which the first operation of a `cache_lock`-protected critical section was to acquire the `slabs_lock`. When `cache_lock` was replaced with a transaction, the transaction immediately would serialize on account of the call to `pthread_mutex_lock`. Since the two locks are also released together at the end of the critical section, in these cases it is correct to change the lock order,

```
1   void func1a() {
2     __transaction_atomic {
3       if (tm_trylock(i.lock))
4         use_item(i);
5         tm_unlock(i.lock);
6       else
7         save_for_later(i);
8     }
9   }
10
11  void func2a() {
12    tm_lock(i.lock);
13    use_item(i);
14    tm_unlock(i.lock);
15  }
```

(a) `func2` privatizes `i`.

```
1   void func1b() {
2     __transaction_atomic {
3       // save_for_later isn't needed, since
4       // func2 no longer accesses i via
5       // privatization.  Instead, this
6       // transaction and line 11 might conflict
7       use_item(i);
8     }
9   }
10
11  void func2b() {
12    __transaction_atomic {
13      use_item(i);
14    }
15  }
```

(b) `func2` does not privatize `i`.

Figure 1: Example of privatization that may arise under less aggressive approaches to transactionalization.

i.e., acquire the `slabs_lock` and then begin a transaction in place of acquiring the `cache_lock`. However, in other cases the `slabs_lock` was only acquired well after the `cache_lock`, and the only way to prevent `cache_lock` transactions from becoming serial and irrevocable was to also replace the `slabs_lock` with transactions.

Another challenge related to per-thread statistics. Over the past few years, many of the statistics counters in memcached have been transformed from global counts to per-thread counters, which are protected by per-thread locks. Unfortunately, *any* operation on a mutex lock is unsafe to perform in an atomic transaction, and thus, we had to replace these locks with transactions. This highlights a flaw with relaxed transactions: when an unsafe operation is performed in a context where conflicts are exceedingly rare, it still necessitates the serialization of all transactions. One solution to this problem would be to make lock operations transaction-safe. Barring such an option, we were forced to replace uncontended per-thread locks with transactions.

Indeed, "transaction-safe" locks were required when replacing the lock governing slab re-balancing. In memcached, a "slab rebalance" lock is used by the cache and slab maintenance threads to prevent concurrent maintenance of the cache and slab data structures. While holding other locks, these threads might use `trylock` calls to determine whether a concurrent maintenance operation is in-flight. To remedy this, we replaced the rebalance lock with a boolean that was modified via transactions. This allowed concurrent transactions to check the state of the lock. While the lock was primarily acquired via a spin loop and `trylock`, in one case it was acquired via a blocking call. Lacking any better alternative, we followed any failed blocking acquire with a call to `pthread_yield`.

Finally, and most unfortunately, we had to replace the item locks, even though they were never contended. While the locking order in memcached is item, cache, slabs, and then stats locks, there are cases in which a maintenance thread attempts to lock an item while holding other locks that come later in the locking order. In these cases, either a spin loop wraps a call to `trylock`, or else a `trylock` is used and failure to acquire results in a handler being registered so that missed items can be processed at a later time (see Line 7 of Figure 1a). While all `item_locks` are acquired using spin loops and `trylock`, we were still faced with an uncomfortable decision: as with the rebalance lock, we could make the lock acquire and release into mini-transactions on a boolean variable, or we could replace every item lock critical section with a transaction. The former choice immediately led to explicit *privatization* [22, 33]: some data protected by item locks would be accessed within transactions, and also outside of transactions (but with the item lock held). We ultimately chose both routes, and developed two branches of the code, one with privatization and the other without.

An illustration of the difference is provided in Figure 1. In branch 'b', where item lock critical sections are replaced with transactions, func1 and func2 may run concurrently and cause conflicts, but `i` is only accessed from within transactions. In branch 'a', where item locks were acquired and released with transactions, func1a is able to inspect the lock, and to use `i` within a transaction as long when the lock is not held, but `i` itself might be accessed outside of a transaction in func2a. Furthermore, the small transaction to acquire the lock in func2a will implicitly take priority over the larger transaction that reads the item's lock in func1a. Note that while branch 'b' is more aggressive with respect to replacing locks with transactions, both branches are correct, since the default TM algorithm in GCC is privatization safe, and this level of safety is a requirement of the Draft C++ TM Specification. Note too that neither option is clearly simpler: branch 'a' required fewer net lines of code to change, but branch 'b' enabled the removal of several corner cases (e.g., the `save_for_later` code path).

```
/* initially mx_can_run = true */

void worker() {
  lock(L);
  do_work();
  if (mx_needed())
    if (!mx_running)
      mx_running = true;
      cond_signal(C); //replace with sem_post(S);
  do_cleanup();
  unlock(L);
}

void halt_maintainer() {
  lock(L);
  mx_can_run = false;
  cond_signal(C);      //replace with sem_post(S);
  unlock(L);
}

void maintainer() {
  while(mx_can_run);
    lock(L);
    do_maintenance();
    unlock(L);
    lock(L);
    mx_running = false;
    cond_wait(L, C);   //replace with unlock(L);
    unlock(L);         //replace with sem_wait(S);
}
```

Figure 2: Comments depict a transformation for removing condition variables used to wake maintenance threads.

3.2 Refactoring Condition Synchronization

The `slabs_lock` and `cache_lock` are used by memcached both to protect critical sections and as the lock object for condition synchronization using `pthread_cond_t` variables. The current Draft C++ TM Specification does not support condition variables, and thus we were required to manually transform all condition synchronization.

A simplifying factor is that condition synchronization in memcached follows two simple patterns. First, there are condition variables for notifying threads when work arrives on a network connection. These are not associated with contended locks, and we did not consider them. The second pattern is for coordinating data structure maintenance. In this pattern, many worker threads can attempt to wake a maintenance thread, which will then modify a data structure. A simplified version of this pattern is depicted in Figure 2. This pattern appears twice, for re-balancing the hash table (via `cache_lock`) and maintaining slabs (via `slabs_lock`).

One flag exists for thread shutdown, and another for determining if the maintainer is currently active. Every attempt to wait on a condition is immediately followed by a lock release, a back edge in the control flow graph, and a lock re-acquire. Furthermore, the code already ensures that there are no spurious wake-ups. Consequently, one can replace the condition variables with semaphores. The changes are trivial, and appear as comments in Figure 2. Rather than call `cond_wait` within a lock-based critical section, maintenance threads call `sem_wait` immediately after completing the critical section. At this stage of the transactionalization, worker threads called `sem_post` from within a lock-based critical section. In stage 3, these critical sections became relaxed transactions, and in stage 5, these calls were moved to `onCommit` handlers within atomic transactions.

```
    lock(stats_lock);
    increment(counter1);
    unlock(stats_lock);
    if (unlikely_condition) {
      lock(stats_lock);
      increment(counter2);
      unlock(stats_lock);
    }
```

Figure 3: Rapid re-locking in memcached.

This transformation resulted in changes to 10 lines of code, not counting comments, and had no impact on performance. While the transformation suffices for memcached, it may still be beneficial for TM to explicitly support condition synchronization (possibilities include conditional critical regions/retry[3, 12], punctuated transactions [32], communicators [19], and transactional condvars [8]). In particular, when `pthread_cond_wait` is not the last instruction in a critical section, and is in a deeper lexical scope, compiler support will be needed to transform the two "halves" of the critical section into separate transactions.

3.3 Maximally Applying the Specification

Having identified the locks requiring replacement, and having developed alternative condition synchronization mechanisms for those locks also associated with condition variables, we were at last able to begin using transactions. We replaced all targeted critical sections with relaxed transactions, maximally applied the callable attribute, and then systematically replaced unsafe operations with safe operations, so that transactions could be marked atomic. Recall that in the absence of transaction cancellation, it would have been correct to leave transactions relaxed after making their operations safe, but that the absence of relaxed transactions guarantees the absence of mandatory serialization.

Replacing Locks, Adding Annotations We began by developing two branches, corresponding to the two approaches to item locks discussed above, in which the `cache_lock`, `slabs_lock`, and `stats_lock` were also transactionalized. This resulted in 51 relaxed transactions in each branch. We then traced all function calls from within relaxed transactions, and marked all functions for which we had the source as callable. This led to 38 annotations in the privatizing itemlocks (IP) branch, and 49 annotations in the transactional itemlocks (IT) branch.

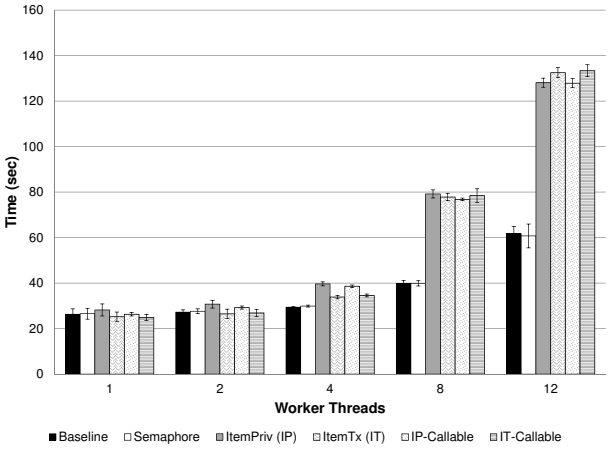

Figure 4: Performance of baseline transactional memcached.

Note that on a few occasions, a function would acquire a lock multiple times within a short region of code (see Figure 3). This pattern does not match with a mental model in which lock acquisitions are expensive. Furthermore, when this entire code region appears within an atomic transaction, transforming each critical section into a nested transaction seems unnecessary. This, in turn, implies that under some circumstances, using TM will encourage programmers to enlarge critical sections.

Figure 4 presents the performance of memcached with semaphores in place of condition variables, and then for each branch with (a) relaxed transactions but no callable annotations, and (b) relaxed transactions and callable annotations. Experiments were performed on a dual-chip Intel Xeon 5650 system with 12 GB of RAM. The Xeon 5650 presents 6 cores/12 threads, giving our system a total of 12 cores/24 hardware threads. The underlying software stack included Ubuntu Linux 13.04, kernel version 3.8.0-21, and an experimental GCC version 4.9.0. All code was compiled for 64-bit execution, and results are the average of 5 trials. Error bars depict a range of +/- one standard deviation.

We generated a workload for memcached using memslap v1.0, downloaded as part of the Ubuntu libmemcached-0.31_1 source package. To ensure that network overheads were not hiding the higher latency of transactions, we ran the memcached server and memslap on the same machine. We ran memslap with parameters `--concurrency=x --execute-number=625000 --binary`. We varied the memslap concurrency parameter (`x`)from 1 to 12 and matched memcached runs with the same number of worker threads plus an additional two maintenance threads. Note that perfect scaling corresponds to an execution time that remains constant at higher thread counts, since each thread executes 625K operations.

In Figure 4, we see that the switch from condition variables to semaphores is negligible, and that privatization (IP) appears to scale better than using transactions in place of item lock critical sections (IT). However, there is no evidence to suggest that the callable attribute (IP-Callable / IT-Callable) improves performance.

Branch	Trans-actions	In-Flight Switch	Start Serial	Abort Serial
ItemPriv (IP)	11201538	625K (5.6%)	625K (5.6%)	10
ItemTx (IT)	3462735	625K (18.0%)	1.25M (36.1%)	0
IP-Callable	10511717	625K (5.9%)	625K (5.9%)	10
IT-Callable	3467927	625K (18.0%)	1.25M (36.0%)	0

Table 1: Frequency and cause of serialized transactions for a 4-thread execution.

Table 1 presents the serialization rates for each branch. Naturally, there are fewer total transactions when transactions replace item locks, since each item lock acquire and release is a separate transaction in the IP branch. However, IP and IT have practically the same number of relaxed transactions that encounter unsafe code on a branch, and must become serial (In-Flight Switch), and that encounter unsafe code on every code path, and must begin in serial mode (Start Serial). A trivial number of transactions abort 100 times in a row, and serialize for the sake of progress.

Handling Volatiles and Reference Counts Many critical sections in memcached access volatile variables as part of periodic maintenance operations or condition synchronization. Strictly speaking, the semantics of such accesses are not defined, though in practice one can expect these volatile variable accesses to be an approximation of C++11's `atomic` types. Accesses to `volatile` and `atomic` variables are both considered unsafe, can not be performed in an atomic transaction, and force transactions to serialize. Therefore, our only hope of achieving scalability was to replace all volatile variable accesses with transactional accesses to non-volatile variables.

This transformation was straightforward. We renamed volatile variables, traced all compilation bugs, and resolved all errors by adding transactions that accessed the new non-volatile variables. In all, we only changed three variables, and the availability of transaction expressions meant that the total lines-of-code count did not change.

However, this transformation raises many questions. First, this transformation was only correct because we manually inspected the code to ensure that concurrent critical sections never interact via volatile variables. Second, memcached only consists of 7400 lines of code, and it is not clear that such a change would be realistic to push through a larger program. Third, GCC currently does not optimize single-location transactions, and thus this change could have a significant impact on performance. Finally, the specification must guarantee that the semantics of transactions are no weaker than the semantics of C++ atomic variable accesses. That is, a transaction expression for reading a variable must have the same ordering guarantees as a read of an atomic variable, and a transaction that sets a variable's

```
1    if (volatile_var == 1)
2       block 1;
3    else if (volatile_var == 2)
4       block 2;
5    else
6       block 3;
```

Figure 5: Re-reading a volatile within a conditional.

value must have at least the same ordering guarantees as a store to a C++ atomic, where in both cases the access uses `std::memory_order_seq_cst`. This is already the case in the Draft C++ TM Specification.

Surprisingly, we found that in some places within memcached, nested `if` statements will re-read the same volatile variable (Figure 5). Based on the assumption that memcached is correct, we did not rewrite these codes.

Memcached uses atomic read-modify-write operations (e.g., `lock incr` on the x86) for reference counting. As with volatile variables, replacing these accesses with transactions was straightforward: we replaced every increment and decrement with a transaction, and replaced every read of these variables with a transaction expression. There was no change to the total lines of code. The main concern with such a transformation is that these assembly operations have memory fence semantics, which must then be guaranteed at transaction boundaries. There is also a question of optimality, since many critical sections increment the reference count, access a datum, and then decrement the reference count. With transactions, it might be possible to replace the modifications of the reference count with a simple read [7].

Performance for this "maximal" transactionalization is presented in Figure 6 and Table 2. For reference, the baseline memcached and "Callable" results are reproduced. At all thread counts, performance degrades, with significant slowdown at high thread counts. Even worse, serialization increased for both branches. In the IP branch, our transformation reduced the number of transactions that started in serial mode, but virtually all of these transactions still ultimately serialized. Such an outcome will consistently hurt performance, since code executes in an instrumented slow path up until the point where serialization is necessary, at which point GCC aborts the transaction and restarts it serially but with less instrumentation. By delaying the point of serialization, we actually hurt performance.

In the IT-Max branch, the transformation was less profitable, and the slowdown more pronounced. Two contributing factors are the increased number of transactions (each of which has higher latency than the code it replaced), and the dramatic increase in transactions that fall back to serial mode due to high abort rates. Since GCC's TM uses direct update [9, 29], the cost of aborts is expected to be high.

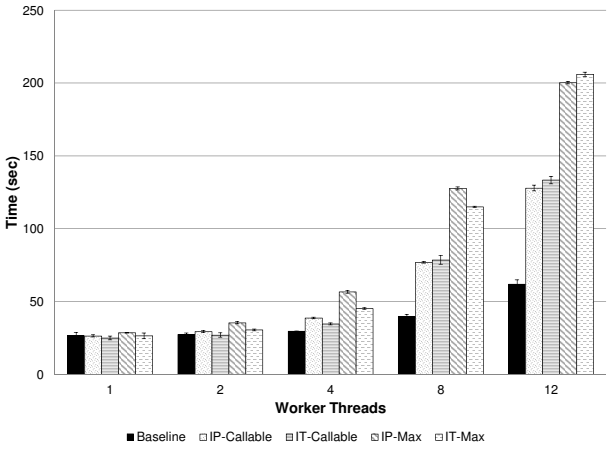

Figure 6: Performance of maximally transactionalized memcached.

Branch	Transactions	In-Flight Switch	Start Serial	Abort Serial
IP-Callable	10511717	625K (5.9%)	625K (5.9%)	10
IT-Callable	3467927	625K (18.0%)	1.25M (36.0%)	0
IP-Max	24085893	1162K (4.8%)	0	87K
IT-Max	6370739	559K (8.8%)	1.25M (19.6%)	66K

Table 2: Frequency and cause of serialized transactions for a 4-thread execution.

3.4 Making Libraries Safe

The most prominent remaining cause of serialization was calls to unsafe standard library functions. These calls fell into a few categories. First, there were operations on untyped memory: `memcmp`, `memcpy`, and `realloc`. Second, there were basic string functions: `strlen`, `strncmp`, `strchr`, `strncpy`, `isspace`, `strtol`, `strtoull`, `atoi`, and `snprintf`. Third, there were calls related to variable arguments: `vsnprintf`, `va_start`, and `va_end`. Lastly, there was a single call to `htons`.

Safety via Reimplementation Most of these functions are simple to transactionalize: we re-implemented the function and marked it transaction safe. This addressed calls to `memcmp`, `memcpy`, `strlen`, `strncmp`, `strncpy`, and `strchr`. Similarly, we re-implemented `realloc` in the naive way, by always allocating a new buffer and using `memcpy`.[1] Unfortunately, the Draft C++ TM Specification requires the transactional and nontransactional versions of a function to be generated from the same source. Thus to make our functions safe, we could not use custom assembly (e.g., vector instructions in memcpy), and thus we had to slow down the non-transactional code path by replacing calls to optimized standard library functions with calls to our naive implementations.

[1] We were able to optimize this slightly, since the initial size of the input is always known in memcached.

```
1   //wrap library function foo inside a
2   //pure function
3   [[transaction_pure]]
4   int pure_foo(char *in, char *out) {
5     return foo(in, out);
6   }
7
8   ...
9   //assume that max sizes of input and output
10  //strings are known otherwise, use malloc
11  //since alloca is not transaction-safe
12  int size = tm_strlen(shared_in_string);
13  char in[size1];
14  char out[size2];
15  //marshall data onto stack
16  for (int i = 0; i < size; ++i)
17    in[i] = shared_in_string[i];
18  //invoke function with non-shared parameters
19  size = pure_foo(in, out);
20  //marshall data off of stack
21  for (int i = 0; i < size; ++i)
22    shared_out_string[i] = out[i];
```

Figure 7: Example of marshaling shared memory onto the stack to invoke an unsafe library function `foo()`.

Safety via Marshaling To make the remaining functions safe, we relied on a novel but unsafe technique. GCC is aggressive about avoiding instrumentation, in particular by noticing when reads and writes are performed to the stack [27] and/or captured memory [6]. By combining this observation with the `transaction_pure` extension, we were able to implement a pattern in which data was marshaled from shared memory onto the stack, so that an unsafe library function could then be invoked using only thread-local parameters. As needed, the result would then be marshaled back into shared memory. A generic example of this approach appears in Figure 7.

Using this technique could be dangerous in buffered update STM algorithms [5], unless the programmer can be sure that the first marshaling operation does not buffer its writes. Although the GCC TM implementation does not use buffered update, we still verified that GCC does not instrument the writes to `in`, or the reads from `out`.

The `isspace`, `strtol`, `strtoull`, and `atoi` functions could all be made safe in this manner, by marshaling the input string onto the stack, calling a wrapped version of the function, and then using the scalar return value without any further marshaling. `htons` did not require any marshaling, since its input and return values are both integers. Similarly, `snprintf` required all its parameters to be marshaled onto the stack, and its output parameter to be marshaled back to shared memory.

Variable Arguments GCC does not yet support variable arguments within transaction-safe functions. Our solution was to manually clone and replace every variable-argument function with a unique version for every combination of parameters that appeared in the program. While this approach is tedious and does not generalize, we believe the effort is valid for a study such as this: there are no performance or correctness reasons why variable arguments will not eventually be transaction safe.

Branch	Transactions	In-Flight Switch	Start Serial	Abort Serial
IP-Callable	10511717	625K (5.9%)	625K (5.9%)	10
IT-Callable	3467927	625K (18.0%)	1.25M (36.0%)	0
IP-Max	24085893	1162K (4.8%)	0	87K
IT-Max	6370739	559K (8.8%)	1.25M (19.6%)	66K
IP-Lib	25658618	625K (2.4%)	0	15K
IT-Lib	8211858	0	625K (7.6%)	10K

Table 3: Frequency and cause of serialized transactions for a 4-thread execution.

Unfortunately, the techniques discussed in this subsection are not general. For example, while one might argue that even after standard libraries become transaction-safe, marshaling would still be useful for third-party libraries, there are many dangers. Chief among them are that a third-party library might at some point begin using transactions internally, resulting in erroneous behavior (especially with buffered update STM), and that marshaling requires, in all cases, that the assembly code be inspected to guarantee that on-stack buffers are being updated appropriately. Another concern is that one must predict the sizes of buffers. In memcached, maximum buffer sizes were easy to discern in all but one location, for which we used a generous 4KB buffer for the input, and 8KB for the output. Furthermore, when a transaction employs the marshaling technique multiple times within a single transaction, the illusion of atomicity may be violated. For example, `snprintf` relies on locale information when handling floating-point values. In a pathological case, the locale could be changed by an administrator between two of these marshaled calls by the same transaction. The output would clearly not appear to be *atomic*. Finally, the absence of side effects in a library is not always certain. A function's implementation might change, introducing static variables or logging that is invisible to the caller.

Figure 8 and Table 3 present the performance when standard library functions were made transaction safe. We see a notable improvement in performance, especially at higher thread counts, though still not as good as the earlier IP-Callable effort. Analyzing the frequency of serialization, we see a marked improvement in all areas: fewer transactions require serialization when they start, fewer become serialized during execution, and fewer abort at a high enough rate to require serialization to ensure progress.

3.5 Moving Code Out of Transactions

Only 6 unsafe functions remained in transactions: `event_get_version`, `assert`, `sem_post`, `fprintf`, `perror`, and `abort`. We began by handling `assert` and

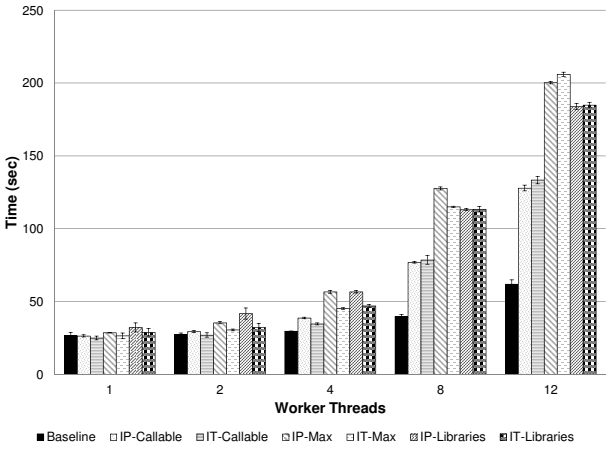

Figure 8: Performance with safe library functions.

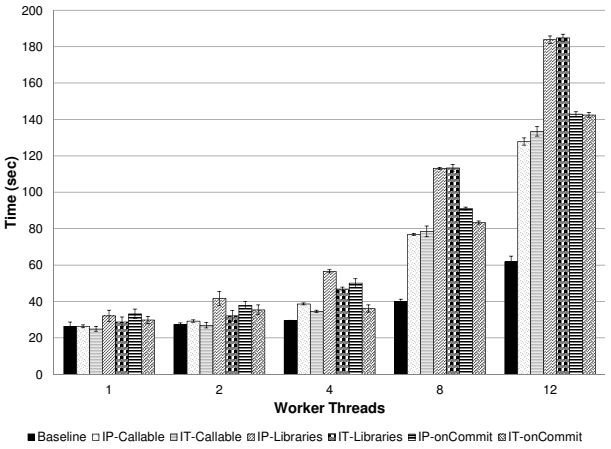

Figure 9: Performance with onCommit handlers.

Branch	Transactions	In-Flight Switch	Start Serial	Abort Serial
IP-Callable	10511717	625K (5.9%)	625K (5.9%)	10
IT-Callable	3467927	625K (18.0%)	1.25M (36.0%)	0
IP-Lib	25658618	625K (2.4%)	0	15K
IT-Lib	8211858	0	625K (7.6%)	10K
IP-onCommit	40305505	0	0	48K
IT-onCommit	8130896	0	0	8K

Table 4: Frequency and cause of serialized transactions for a 4-thread execution.

`abort`, which can immediately terminate a program, without calling any `atexit` functions. If the underlying TM is opaque [11], then at the point where the assertion evaluates to false or the `abort` is encountered, there exists an equivalent lock-based execution in which termination would be justified. Furthermore, since `atexit` functions are not called, other threads should not be able to observe the intermediate state of the transaction calling `assert` or `abort`. Consequently, it is safe to simply terminate the program at these points. To achieve this effect, we wrapped the assert/abort code and accompanying I/O in a pure function. Note that the I/O only involved string literals. Note, too, that while this change allowed many relaxed transactions to be marked as atomic, there was no impact on the frequency of serialization, since the relevant code never runs in a correct execution of memcached.

To handle `event_get_version`, we assumed that the version of `libevent` would not change during program execution. We then called the function outside of a transaction, and used the stored value in place of a function call.

Finally, we removed remaining `fprintf` (which log certain events to `stderr` when a program-wide flag is set), `perror`, and `sem_post` calls by registering `onCommit` handlers. In GCC, these handlers all run after the respective transaction commits *and releases all locks*. That being the case, our use of `onCommit` handlers has the potential to produce output in a different order than a lock-based program, since I/O performed by the handler does not complete atomically with the associated critical section. Furthermore, in the case of `perror`, we could not simply delay the function, but instead saved the `errno` and then called `strerror_r` in the commit handler. Unlike I/O, there are no concerns about ordering when delaying `sem_post` via an `onCommit` handler, since the only uses of condition synchronization are to wake up maintenance threads.

The only other challenge when using commit handlers was that some code could be called from both transactional and nontransactional contexts. GCC's TM does not expose the function for checking if a thread is in a transaction. We made this function visible to the program, in order to check for cases when the onCommit handler should be registered, versus those times when the handler should run immediately.

The performance following these changes took a remarkable turn. As shown in Figure 9, running times dropped to almost as low as the previous best, the simple IP-Callable branch. More importantly, when there is no mandatory serialization, transactionalization of item locks performs better than the use of privatization. Table 4 verifies that transactions no longer serialize at begin time, or due to an unsafe call during their execution. With all unsafe operations removed, the impact of serialization due to aborts becomes the dominant factor distinguishing the two approaches to item locks. The transactional version, which does not prioritize the small lock acquire/release transactions of the IP branches, has fewer serializations, leading to lower execution times.

4. Modifying GCC-TM

In the previous section, we eliminated *all* relaxed transactions, to include those that performed logging (via `fprintf` and `perror`) and those that ultimately would terminate the program (via `assert` and `abort`). Throughout this paper, we have claimed that atomic transactions provide a perfor-

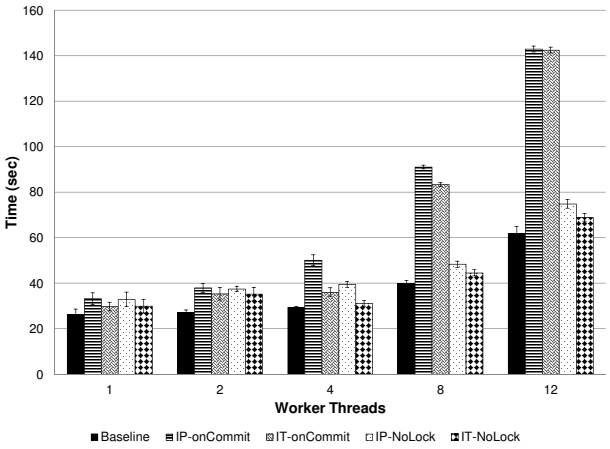

Figure 10: Performance without the readers/writer lock.

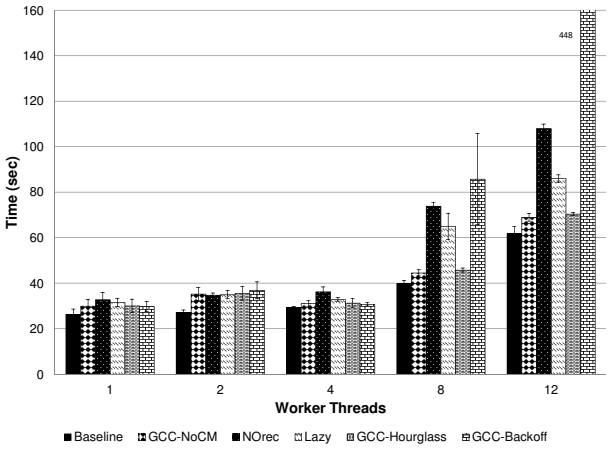

Figure 11: Comparison to other TM algorithms and contention managers.

mance model to the programmer. To illustrate the significance of this model, we made two further changes.

The GCC implementation of TM assumes that serialization is a common occurrence. To make serialization cheap, all transactions acquire a single global readers/writer lock in read mode when beginning a transaction, and release it on commit or abort. Transactions that require serialization acquire the lock in write mode. The availability of this lock makes switching to serial mode cheap, simplifies corner cases related to thread creation and joining, and enables a simple contention management policy, wherein a transaction becomes serial after 100 consecutive aborts.

The single readers/writer lock is an obvious bottleneck, but for programs with even a single relaxed transaction, serialization may be required for correctness. This holds true even with other contention managers [30]. Once we removed the last relaxed transaction, we removed the readers/writer lock from the GCC TM library, and added a separate lock exclusively for thread creation/joining. We then added a variety of simple contention managers (exponential backoff [14], a modified form of serialization called "hourglass" [10, 18], or simply no contention management), as well as a "lazy" STM algorithm[2] and the NOrec STM algorithm [4].

Figure 10 repeats the IT and IP results with onCommit handlers, and adds two new curves for GCC without the readers/writer lock. In these curves, there is no contention management: transactions immediately retry until they succeed. At high thread counts, we now can see that contention on the readers/writer lock is the primary source of overhead. Furthermore, performance is within 30% of the baseline memcached. This is despite expensive instrumentation (every read and write of shared data involves a function call, and every transaction boundary involves the creation of a checkpoint), and is in comparison to a highly optimized lock-based baseline.

[2] This algorithm uses the same lock table as the default GCC algorithm, but buffers updates and acquires locks at commit time

In Figure 11, we consider only the best-performing of these algorithms, IP-NoLock, which we now call "GCC-NoCM". We compare it to the Lazy and NOrec STM algorithms, also configured without contention management, as well as versions of the GCC STM algorithm using backoff or the hourglass contention manager (configured to prevent new transactions after 128 consecutive aborts).

With regard to contention managers, we see that the hourglass performance best matches performance without any contention management, making it an interesting candidate for cases where livelock is not expected, but programmers desire some guarantee of forward progress. At high thread counts, backoff performs poorly due to preemption. Furthermore, since the workload is heterogeneous, with dozens of source-code transactions of varying sizes and access patterns, backoff is not an optimal choice even at lower thread counts.

Similarly, we found that the GCC algorithm, which does not have buffered update, had the lowest latency and the best scalability. This is despite extremely high abort rates: at 12 threads, NOrec worker threads aborted once per 5 commits, Lazy worker threads aborted 14 times per 1 commit, and GCC worker threads aborted 12.6 times per 1 commit. In the case of NOrec, the frequency of small writer transactions induced a bottleneck on internal NOrec metadata; for both NOrec and the Lazy algorithm, the need to buffer byte-by-byte stores in `memcpy` and then read them later as words necessitated an expensive logging mechanism. Additionally, the variance in abort rate among threads was an order of magnitude lower for GCC than for Lazy, suggesting that Lazy was more prone to bouts of starvation.

These results suggest that real workloads do exhibit sensitivity to contention management and STM algorithm decisions, that real workloads place stresses on TM algorithms that research prototypes often ignore, and that expert programmers should be able to tune these parameters.

5. Recommendations for System Designers

We believe that the most viable path forward is for programmers to strive to avoid serialization, and system developers to both (a) optimize for the serialization-free case, and (b) facilitate the creation of programs that do not require serialization. To this end, we offer the following recommendations to designers of transactional runtime libraries, and to the authors of future TM language specifications.

Implementors Should Assume Serialization is Rare The ultimate success or failure of transactional programming will depend on performance. Serialization of relaxed transactions is a dangerous obstacle. The GCC TM library assumes serialized relaxed transactions are common, and optimizes that case at the expense of workloads in which non-serializing atomic transactions abound.

To be clear, relaxed transactions are necessary: they are the only way to perform I/O using transactional data, and in the absence of I/O, relaxed transactions should perform on par with atomic transactions. When transactionalizing legacy C programs, the only justification for atomic transactions is the static guarantee that they will not force serialization. While it ought to be possible to approach the performance in Figure 10 with a different lock implementation [17], even a scalable lock will be a bottleneck: if many transactions must serialize, then the serial fraction of the program will be too high to achieve good performance. It is probably better to increase overhead for the serial transaction to avoid bottlenecks [35, 36]. Note, too, that the latest version of GCC requires every *hardware* transaction to use this lock, suggesting that hardware TM will not achieve its full potential as long as serialized transactions are the common case.

The Specification Must Address Condition Synchronization The C++ Draft TM Specification does not allow a lock to be replaced with a transaction if that lock is also used in conjunction with a condition variable. In this work, we were able to leverage the specific communication pattern between threads, and replace condition variables with semaphores. Our technique does not generalize, and relies on the availability of `onCommit` handlers. Given the widespread use of condition variables in real-world programs, it is essential that the specification provide a solution. Otherwise, TM adoption will remain limited.

Specify More Transaction-Safe Libraries Serialized relaxed transactions are currently the only mechanism by which calls to standard library functions are possible. Our ad-hoc work-arounds, while necessary for evaluating the specification, do not generalize, and are not safe in the common case. Programmers should not develop ad-hoc approaches, marshal parameters into private memory, or use `transaction_pure`. The programming environment should also provide support for managing terminating exceptions (e.g., asserts), so that programmers can continue to develop robust, self-diagnosing code while using transactions.

Furthermore, transaction safety should not inhibit nontransactional performance. In the current specification, a single function body is used to produce two code "clones" appropriate for transactional and nontransactional uses, and thus transaction-safe code cannot include inline assembly. In the long term, application programmers and system developers alike will benefit from the ability to provide different implementations of a function depending on the calling context.

OnCommit Handlers Should Be Part of the Specification Our experience greatly benefited from `onCommit` handlers, which were removed from the 1.1 Draft Specification. Their return to the specification will allow programmers to avoid serialization, and to move non-critical code (i.e., logging) out of critical sections. We are less convinced that `onAbort` handlers are needed: the only role we envisioned for them in memcached was to employ backoff after a failed transaction. Specifying a simple contention manager interface is likely more appropriate.

Ignore Further Ease of Use Concerns Finally, we believe that the developers of transactional environments should not worry too much about ease-of-use. In particular, we found that manual annotation of safe and callable functions, while tedious, was not difficult. Furthermore, it was not error-prone, since incorrect `transaction_safe` annotations immediately generated compiler errors. Similarly, being forced to re-implement volatile variables, locks, and condition synchronization was unpleasant, but ultimately rewarding. We discovered, for example, that replacing item locks with transactions removed corner cases, and indeed there are several optimizations to memcached that are now possible on account of transactional reference counts and the elimination of delayed maintenance code paths. If locks and volatiles were transaction-safe, we would not have identified these opportunities.

6. Recommendations for Programmers

The success of transactional programming will depend on cooperation between the programmer and system developer, where both work together to avoid serialization. Our recommendations to application developers appear below:

Performance Model While there are semantic differences between relaxed and atomic transactions, for legacy code the key difference relates to the performance model exposed to the programmer. When a transaction is atomic, the programmer can be sure that there is no artificial obstacle to concurrency, whereas a relaxed transaction might contain some operation (to include a seemingly benign call to a variable argument function) that forces serialization. For this reason, we believe that programmers should rewrite code to avoid relaxed transactions whenever possible.

Our study entailed only one application, written in C, and did not use any of the exception-related features of the Draft C++ TM Specification, which apply only to atomic transactions. While it would be premature to draw any conclusions about the need for both atomic and relaxed transactions, we nonetheless feel that both play an important role. In our view, atomic transactions carry a performance model that is statically checked, and about which the programmer can reason. Even when performance does not match the model, the fact that atomic transactions were used guides the programmer's analysis: some potential problems, such as mandatory serialization, simply are not possible.

On the other hand, relaxed transactions provide a safety guarantee for operations that cannot be atomic, at the expense of a performance model. The best case performance for relaxed transactions should match atomic transactions, but without the compiler's guidance, we would not have been able to achieve any confidence that our use of transactions would avoid serialization. Relaxed transactions play a necessary role in allowing I/O with transactional data, calling libraries that are not yet instrumented for transactions, and allowing transactions to interact with nontransactional threads. Programmers must be savvy enough to know when a relaxed transaction is the right choice.

Incremental Transactionalization is a Myth Our initial goal was to replace `cache_lock`. However, we had no choice but to expand our scope to include an array of item locks, as well as every major lock in the program, whether it was contended or not. When transactionalizing legacy code, we expect this experience to be the norm: if the last lock in a lock hierarchy is the only contended lock, then either the program lacks concurrency, or there is a trivial technique for splitting that lock into a set of low-contention locks. It is far more likely that some intermediate lock in the hierarchy will be contended, requiring the programmer to replace many locks with transactions.

While any transactionalization will entail considerable effort, it resembles a refactor more closely than a redesign. While we may be overly optimistic, we found that the tedious process of annotating functions and replacing locks and volatile variables provided an opportunity to identify optimizations in a number of areas (reference counting, interaction with the maintenance threads, statistics reporting), which we intend to improve as future work.

Expect Limited Tool Support Manually diagnosing the causes of aborts and serialization in our program was challenging, and we eventually extended the GCC TM library to use the Linux `execinfo.h` infrastructure to provide more meaningful profiling data. While static information about serialization could be generated at compile time, and dynamic information about aborts at run time, we did not see the absence as unreasonable. These tools will arrive eventually, but system developers' effort should first be placed on making standard libraries safe, and condition synchronization possible.

Similarly, tools for automatically annotating functions do not seem particularly valuable. To achieve good performance, we had to modify most source files in memcached. That being the case, the additional task of annotating functions was not overly burdensome. We expect tools to eventually make this task easier, but in our opinion it is not urgent.

Expect to Fork the Code Ultimately, we believe that the annotations and transformations we made to memcached are too complex and widespread to hide behind macros. Currently, memcached compiles on many operating systems and platforms, to include those that lack compiler support for atomic operations (e.g., `lock inc`), with macros hiding platform-specific implementation differences. The need for cross-platform support can be an obstacle to progress: it is even unlikely that memcached `volatiles` will be replaced with C++ `atomic` data types any time soon, and if such a conversion occurs, it will likely be handled via even more macros. Preserving multi-platform support while performing the modifications discussed in this paper would create inscrutable, un-maintainable code.

7. Conclusions

In this paper, we presented our experiences transactionalizing the memcached in-memory web cache using GCC. Our focus was analyzing the Draft C++ TM Specification, not on evaluating a particular TM mechanism or tool. Nonetheless, our version of memcached is a valuable benchmark, which we will release as open source code.

Our main findings related to the cost of unintended serialization, the effort required to avoid serialization, and the contract between system developer and programmer that is most likely to result in a viable transactional programming environment. Throughout the paper we identified areas where the specification could be changed to improve programmability, transformations and analyses that programmers will need to perform, opportunities for tool creators, and maintenance and performance challenges that will arise during the piecemeal transactionalization of multi-platform programs.

We hope that this study will assist standardization efforts by providing further insight into the challenges that will be faced when transactionalizing legacy code, and that our source code will provide TM algorithm designers with a new workload for evaluating implementations.

Acknowledgments

We thank our anonymous reviewers, whose detailed suggestions greatly improved this paper. We also thank Justin Gottschlich, Victor Luchangco, Jens Maurer, and Torvald Riegel for many helpful conversations about the Draft C++ TM Specification.

References

[1] A.-R. Adl-Tabatabai, T. Shpeisman, and J. Gottschlich. Draft Specification of Transactional Language Constructs for C++, Feb. 2012. Version 1.1, http://justingottschlich.com/tm-specification-for-c-v-1-1/.

[2] M. Ansari, C. Kotselidis, K. Jarvis, M. Lujan, C. Kirkham, and I. Watson. Lee-TM: A Non-trivial Benchmark for Transactional Memory. In *Proceedings of the International Conference on Algorithms and Architectures for Parallel Processing*, June 2008.

[3] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An Object-Oriented Approach to Non-Uniform Cluster Computing. In *Proceedings of the 20th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, San Diego, CA, Oct. 2005.

[4] L. Dalessandro, M. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, Jan. 2010.

[5] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[6] A. Dragojevic, Y. Ni, and A.-R. Adl-Tabatabai. Optimizing Transactions for Captured Memory. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, Calgary, AB, Canada, Aug. 2009.

[7] A. Dragojevic, M. Herlihy, Y. Lev, and M. Moir. On The Power of Hardware Transactional Memory to Simplify Memory Management. In *Proceedings of the 30th ACM Symposium on Principles of Distributed Computing*, San Jose, CA, June 2011.

[8] P. Dudnik and M. M. Swift. Condition Variables and Transactional Memory: Problem or Opportunity? In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Raleigh, NC, Feb. 2009.

[9] P. Felber, C. Fetzer, and T. Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[10] F. E. Fich, V. Luchangco, M. Moir, and N. Shavit. Obstruction-free Algorithms Can Be Practically Wait-free. In *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sept. 2005.

[11] R. Guerraoui and M. Kapalka. On the Correctness of Transactional Memory. In *Proceedings of the 13th ACM Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

[12] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable Memory Transactions. In *Proceedings of the 10th ACM Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[13] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[14] M. P. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing*, Boston, MA, July 2003.

[15] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. Eigenbench: A Simple Exploration Tool for Orthogonal TM Characteristics. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Atlanta, GA, Dec. 2010.

[16] G. Kestor, S. Stipic, O. Unsal, A. Cristal, and M. Valero. RMS-TM: A Transactional Memory Benchmark for Recognition, Mining and Synthesis Applications. In *Proceedings of the 4th ACM SIGPLAN Workshop on Transactional Computing*, Raleigh, NC, Feb. 2009.

[17] Y. Lev, V. Luchangco, and M. Olszewsk. Scalable Reader-Writer Locks. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures*, Calgary, AB, Canada, Aug. 2009.

[18] Y. Liu and M. Spear. Toxic Transactions. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, San Jose, CA, June 2011.

[19] V. Luchangco and V. Marathe. Transaction Communicators: Enabling Cooperation Among Concurrent Transactions. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, San Antonio, TX, Feb. 2011.

[20] D. Lupei, B. Simion, D. Pinto, M. Misler, M. Burcea, W. Krick, and C. Amza. Transactional Memory Support for Scalable and Transparent Parallelization of Multiplayer Games. In *Proceedings of the EuroSys2010 Conference*, Paris, France, Apr. 2010.

[21] N. Mathewson and N. Provos. Libevent – An Event Notification Library, 2011–2013. http://libevent.org/.

[22] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[23] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*, Seattle, WA, Sept. 2008.

[24] V. Pankratius and A.-R. Adl-Tabatabai. A Study of Transactional Memory vs. Locks in Practice. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, San Jose, CA, June 2011.

[25] L. Poettering. Measuring Lock Contention, 2009–2013. http://0pointer.de/blog/projects/mutrace.html.

[26] M. Pohlack and S. Diestelhorst. From Lightweight Hardware Transactional Memory to Lightweight Lock Elision. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, San Jose, CA, June 2011.

[27] T. Riegel, C. Fetzer, and P. Felber. Automatic Data Partitioning in Software Transactional Memories. In *Proceedings of*

the 20th ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 2008.

[28] C. Rossbach, O. Hofmann, and E. Witchel. Is Transactional Programming Really Easier? In *Proceedings of the 15th ACM Symposium on Principles and Practice of Parallel Programming*, Bangalore, India, Jan. 2010.

[29] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System For A Multi-Core Runtime. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming*, New York, NY, Mar. 2006.

[30] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[31] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, Ottawa, ON, Canada, Aug. 1995.

[32] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with Isolation and Cooperation. In *Proceedings of the 22nd ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, Montreal, Quebec, Canada, Oct. 2007.

[33] M. Spear, V. Marathe, L. Dalessandro, and M. Scott. Privatization Techniques for Software Transactional Memory (POSTER). In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, Portland, OR, Aug. 2007.

[34] M. Spear, M. M. Michael, and M. L. Scott. Inevitability Mechanisms for Software Transactional Memory. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Transactional Computing*, Salt Lake City, UT, Feb. 2008.

[35] M. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott. Implementing and Exploiting Inevitability in Software Transactional Memory. In *Proceedings of the 37th International Conference on Parallel Processing*, Portland, OR, Sept. 2008.

[36] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable Transactions and their Applications. In *Proceedings of the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[37] F. Zyulkyarov, V. Gajinov, O. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. Atomic Quake: Using Transactional Memory in an Interactive Multiplayer Game Server. In *Proceedings of the 14th ACM Symposium on Principles and Practice of Parallel Programming*, Raleigh, NC, Feb. 2009.