# Thrifty-malloc: A HW/SW Codesign for the Dynamic Management of Hardware Transactional Memory in Embedded Multicore Systems

Thomas Carle[1,4], Dimitra Papagiannopoulou[1], Tali Moreshet[2], Andrea Marongiu[3], Maurice Herlihy[1], and R. Iris Bahar [1]

[1]Brown University, Providence, RI, USA, {name_surname@brown.edu}
[2]Boston University, Boston, Massachusetts, USA, {talim@bu.edu}
[3]University of Bologna, Bologna, Italy, {a.marongiu@unibo.it}
[4]University of Toulouse, Toulouse, France, {thmscarle@gmail.com}

## ABSTRACT

We present *thrifty-malloc*: a transaction-friendly dynamic memory manager for high-end embedded multicore systems. The manager combines modularity, ease-of-use and hardware transactional memory (HTM) compatibility in a lightweight and memory-efficient design. Thrifty-malloc is easy to deploy and configure for non-expert programmers, yet provides good performance with low memory overhead for highly-parallel embedded applications running on massively parallel processor arrays (MPPAs) or many-core architectures. In addition, the transparent mechanisms that increase our manager's resilience to unpredictable dynamic situations incur a low timing overhead in comparison to established techniques.

## 1. INTRODUCTION

Each generation of embedded systems provides more and more processing cores, and it remains a challenge to provide application programming interfaces (APIs) that facilitate safe and effective programing. For shared-memory embedded systems, *dynamic memory management* is particularly important. Modern object-oriented languages require the ability to create objects in a way that cannot be predicted statically. Because embedded systems are frequently resource-constrained, static, conservative over-allocation is not feasible. Instead, such systems require a lightweight, dynamic memory manager, which must be implemented without relying on an underlying operating system.

For multithreaded and parallel/distributed applications, *locks* have been historically used as synchronization primitives. Nevertheless, it is known that lock performance scales poorly, and locking has many pitfalls, especially for non-expert programmers. In particular, deadlocks may be disastrous for embedded applications. *Hardware transactional memory* (HTM) [13] is a more programmer-friendly synchronization alternative. It optimistically allows critical sections to execute in parallel, providing built-in recovery mechanisms if data conflicts do occur. Unfortunately, conventional dynamic memory managers[6] are usually not compatible with HTM mechanisms: calls to *malloc()* and *free()* must take place outside of transactions, complicating the programmer's task, often requiring static over-allocation. Moreover, the few memory managers that are compatible with HTM[15, 8] are not well-suited to resource-constrained embedded systems.

To overcome these problems, we propose a general method to make dynamic memory management compatible with both embedded systems and HTM, with the goal of providing both performance scalability and ease of use. The programmer calls *malloc()* and *free()* either from inside or outside transactions. For better performance, this method minimizes the number of synchronization operations by provisioning a small local (private) memory pool to each thread. Under normal conditions, a thread allocates from and frees to this pool without synchronization. Under exceptional conditions, a thread may exhaust its local pool, falling back to a transparent runtime mechanism which automatically allocates additional memory to the threads that need it. This service is performed using transactions that are dedicated to this purpose only, and are clearly separated from any application-level transactions specified by the programmer. The only *a priori* difference between using local pools and allocating directly from the heap is that allocating directly from the heap offers more flexibility in the use of the memory (none of it is reserved for any given thread). However this flexibility comes with a temporal cost: threads have to synchronize everytime they want to perform a malloc.

We call our efficient dynamic memory manager *Thrifty malloc*. Thrifty malloc can be used with locks or HTM and is particularly suitable for embedded multicore systems. This paper makes the following contributions:

- Thrifty malloc is lightweight: it does not depend on an underlying operating system.
- Thrifty malloc is tunable. It works correctly without programmer intervention, but offers a simple tuning interface that can be adapted to specific applications.
- We compare thrifty malloc to two alternatives: managing memory entirely outside of transactions, and using locks to protect the heap. Experimental results show that in most cases, thrifty malloc outperforms or matches these alternatives.

The remainder of the paper is organized as follows: Section 2 contains a presentation of the related work, Section 3 presents our method, its algorithms and our underlying architecture. Section 4 presents the results obtained using our



*CASES '16, October 01-07 2016, Pittsburgh, PA, USA*



ISBN 978-1-4503-4482-1/16/10...$15.00

DOI: http://dx.doi.org/10.1145/2968455.2968513

method on representative benchmarks, and finally Section 5 concludes.

# 2. BACKGROUND AND RELATED WORK

## 2.1 Dynamic memory management

Most dynamic memory management systems use a shared data structure, shared by all the threads, to represent available memory blocks [26]. Each time a thread allocates or frees memory, it must synchronize access to the shared data structure with any concurent threads. Such frequent synchronization can degrade performance, especially for threads that require substantial dynamic memory management.

One common technique to alleviate such synchronization overheads is to split memory into multiple pools, allowing requests to distinct pools to be handled in parallel. Pools may be thread-private or shared. In turn, each pool can be divided into *bins* (subpools), each bin managing blocks of one particular size. A thread allocating a block actually allocates the smallest block of size greater than or equal to the size requested. Although bins can waste memory, most existing memory management systems are designed to optimize speed. Dynamic memory managers employing bins include Hoard [6], TCmalloc [11] and others [19].

Many embedded systems, however, are memory-limited. As a result, the number and size of the pools requires a tradeoff between the quantity of memory available in the system, the desired degree of parallelism, and the simplicity of the memory manager design. Thrifty-malloc is designed with all these constraints in mind.

## 2.2 Hardware Transactional Memory

Shared-memory multicores need a way to synchronize concurrent access to shared data structures. Usually, concurrent threads synchronize via locks. Locks, however, can slow performance and consume excessive energy, as they typically require energy-expensive read-modify-write operations that traverse the memory hierarchy. In addition, locks must be deployed conservatively whenever conflicts are possible, even if they are very unlikely. By contrast, *speculative* synchronization mechanisms detect conflicts dynamically, rolling back and retrying computations when conflicts actually occur, instead of delaying computations that might encounter them.

Transactional memory [13, 23], is a speculation technique that allows critical sections to execute in parallel. If a data conflict does take place, it is detected, and one or more of the conflicting threads is rolled back and restarted. When designing hardware speculative synchronization mechanisms for embedded devices, it is essential to keep both the underlying hardware and the software interface simple and scalable. There are three main components required to implement hardware transactional memory: transactional bookkeeping, data versioning, and conflict management/resolution. Transactional bookkeeping tracks transactional readers and writers. For each data line, there can be multiple transactions reading that line, or a single transaction writing it. If a transaction attempts to access (i.e., read or write) a data line that has been modified (i.e., written) by another transaction, a data conflict occurs. Data versioning keeps track of speculative and non-speculative versions of data. That is, if a transaction modifies data, the old non-speculative data must be saved somewhere so it can be resorted in the event of a conflict. Finally, the conflict management and resolution scheme is responsible for initiating the recovery process, selecting the transactions that must be aborted, have their original data restored, and restart their execution from the beginning. Repeated aborts can be very costly both in terms of runtime and power overhead so care must be taken to minimize the conflict rate.

In conventional architectures, locks are memory locations manipulated by test-and-set (or similar) operations. In the architecture used here, locks reside is a dedicated test-and-set memory distinct from main memory. In either case, locks should not be manipulated inside transactions. In conventional architectures, such as Intel's Haswell [16], test-and-set operations are forbidden within transactions, and will cause the containing transaction to abort. In our architecture, rolling back a transaction would not release any locks it might have acquired.

## 2.3 Allocating memory within transactions

Since the combination of transactional memory and locks can be dangerous, classical allocators that rely on locks to synchronize cannot be used inside transactions. To overcome this problem, allocators that are compatible with transactional memory were developed. The basic idea is to use lock-free algorithms in order to perform synchronizations, or to include special kernel drivers to monitor the execution and take actions when necessary.

*McRT-Malloc.*

Hudson *et al.* present a scalable software transactional memory allocator called McRT-Malloc [15]. The heap is divided into superblocks that the threads can request and own. These superblocks are used as memory bins, containing only sub-blocks of a certain size that threads can use to serve the *malloc()* requests. Superblocks are organized using two lists: one private list from which the thread owning the superblock can allocate, and one public list to which concurrent threads can return the sub-blocks that were originally allocated from the superblock. When there are no more blocks to allocate in the private list, the owner thread tries to repatriate the blocks in the public list to the private list using a non-blocking algorithm. Although allocating from the private list requires no synchronization, returning blocks to the public list is performed via concurrent accesses, and thus requires synchronization. This memory freeing process is performed by non-blocking algorithms based on the Compare-And-Swap primitive (CAS). Consequently, this method requires hardware support under the form of an efficient CAS primitive. The algorithm is designed in order to minimize the use of the CAS primitive, and implements a protocol which reuses freed bits in order to efficiently avoid the ABA problem[14][1]. Overall, this allocator displays good performance scaling on the tested benchmarks. However this solution is subject to some limitations that may make it difficult to use in embedded systems:

- This allocator does not allow concurrent allocation between transactional threads and non-transactional ones: in other words, a thread cannot call *malloc()* from outside a transaction unless this call is statically guaranteed to never happen concurrently with a call from inside a transaction. We want our solution to allow the transparent use of *malloc()* from both inside and

[1]The ABA problem: a thread reads a value A from a shared location, other threads change the location to value B, and back to A again (but with other parts of the data structure modified), causing a CAS to succeed when it shouldn't.

outside transactions at the same time, in order to simplify the programmer's work.

- As we described, the method relies on dividing the heap into superblocks which are themselves divided into memory bins. This design was created with average-case performance as the only optimization criterion, and does not take into account the particular constraints of embedded systems. As a consequence, this memory allocation strategy is not naturally conservative in terms of overall memory requirements. Enforcing memory efficient behavior for any particular application would require individual tuning and configuration, and in turn, experienced programmers with good knowledge of the behavior of the embedded applications. This means increased development costs, increased time-to-market, and potentially more bugs. For embedded systems, a simpler even if less efficient mechanism is preferable.

  Our allocator trades some performance for ease-of-use (in particular it can be used on different applications without reconfiguration) and a simple allocation strategy implementing a general memory-efficient behavior.

- It requires some standard Operating System (OS) services, in particular for transactional memory virtualization. We want our solution to provide services that do not rely on an OS, since many embedded systems execute *baremetal* code, i.e. without an OS running.

### *LFMalloc.*

In LFMalloc[8], the allocator divides the heap into superblocks allocated to each processing core. The objective is to find a tradeoff between totally centralized structures that perform poorly for parallel applications, and thread-local pools that use a lot of memory as the number of threads increases. The authors report very good performance scaling of their allocator, especially when only one thread is executed per core: since the memory pools are allocated to the cores, no extra synchronization is necessary when only one thread executes on each core. This setup is compatible to our target embedded execution platform, where only one thread is allowed to execute on each core. However, LFMalloc was designed for a complex multithreaded environment, featuring thread migration and preemption. The allocator is similar to Hoard[6], utilizing superblocks as in McRT-Malloc.

The main differences between these allocators are:

- In LFMalloc, the superblocks are allocated on a per-core basis instead of on a per-thread basis, which makes the memory requirements independent from the number of threads.

- The LFMalloc algorithm uses a lock to synchronize threads when they return memory to the public list of a superblock, while McRT-Malloc and Hoard use CAS-based lock-free algorithms.

- To deal with scheduling events such as preemptions and migrations, which make the use of a lock hazardous, LFMalloc relies on a special kernel driver[24]. Should such an event occur while a thread is executing the critical section of the allocator algorithm, the driver would detect it and act as a transactional abort mechanism.

To overcome this last limitation, LFMalloc has been implemented using HTM (with lock elision) for its own synchronization[10]. Nevertheless, this does not make this implementation directly compatible with system-level HTM synchronization, since nested transactions may not be supported by the hardware. Moreover, as with McRT-Malloc, the division of the available memory into superblocks and bins can be problematic in the particular context of embedded systems.

### *CIA-Malloc.*

In [9], the authors address the particular problem of cache conflict misses that can be induced by memory allocators. On architectures featuring a data cache, allocating blocks without considering where they are mapped in the L1 cache may result in an unbalanced repartition of the active blocks over the cache indices. This situation increases the miss conflict rate in the cache and incurs timing penalties. In systems featuring HTM, these miss conflicts can trigger unnecessary aborts that are even more expensive. Cache-Index Aware Malloc (CIA-Malloc) [4] was proposed to tackle the conflict misses problem. As with LFMalloc, it is based on local pools allocated to each core, and divided into superblocks acting like memory bins. Since all sub-blocks of a superblock have the same size, the repartition of their addresses usually follows a regular pattern. This regularity can limit the number of cache indices corresponding to the sub-block addresses. In order to distribute the sub-blocks to a wider range of indices, CIA-Malloc features a particular organization of the superblocks: some sub-block addresses are shifted by the size of a cache line (and thus correspond to a new index in the cache), by making ranges of addresses non allocatable. This technique trades some memory to significantly reduce the conflict miss rate compared to index-oblivious allocators, and avoids pathological situations in the presence of HTM. Although this issue must be taken seriously for architectures featuring a data cache, we do not consider it in the scope of this paper. Our method is general and could be adapted without difficulty to take into account cache indices, but in the context of our work presented in this paper we target the cacheless architecture of Section 3.2 in which this particular problem does not occur.

### *Software Transactional Memory related issues.*

Baldassin *et al.* present a similar problem that can arise when working with software transactional memory (STM) [5]. Indeed, one possible design for STM is to use locks to emulate the transactional behavior. In order to keep the number of locks relatively low, the memory space is divided into areas called *stripes*, each of which is protected by a lock. Consequently, enlarging the stripes reduces the number of required locks. However, it also increases the probability that two memory addresses accessed by different transactions will be mapped to the same stripe. In this case, the first transaction to access a memory location in the stripe will acquire the lock and access the memory. If a second transaction tries to access another memory location in the stripe before the first transaction ends, it will be unable to acquire the lock and will be aborted as a result. This kind of abort is referred to as *false abort*. This situation makes the abort rate of transactions dependent on the memory allocation policy: depending on how and where the allocated blocks are picked, false aborts will be more or less likely to occur. Although this is an important problem, in the scope of our work we limit ourselves to HTM, for which the trans-

actional mechanisms are part of the hardware design and are thus not emulated using locks. As a result, the placement of the allocated memory blocks cannot result in false aborts.

# 3. THRIFTY-MALLOC IMPLEMENTATION

In this section, we present the principles of our method in more detail along with some implementation features of our memory manager.

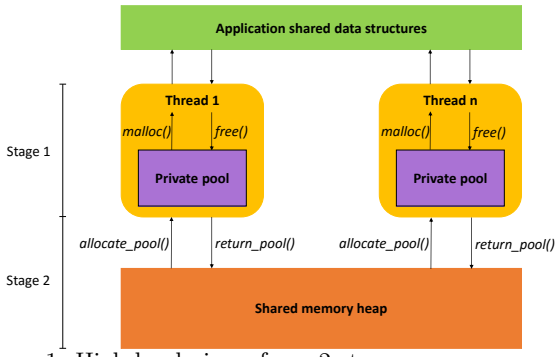



Figure 1: High-level view of our 2-stages memory manager

## 3.1 Allocation/deallocation principles

Our memory manager implements a two-level scheme, as depicted in Figure 1; each thread is allocated a *private memory pool* from the heap at system startup, from which it can directly allocate and return memory blocks at runtime. Using private pools greatly reduces the number of synchronizations the threads have to perform, and is thus adapted to parallel applications. The private pools allocated to the threads at system startup do not cover the entirety of the heap. Most of the memory remains unallocated, and can be used to refill the local pools of the threads, should they run out at run-time. Using this approach, we preserve the dynamicity of the applications, by letting the threads request memory if and when they need it, instead of statically dividing all the memory prior to the execution.

Our memory manager follows a first fit strategy [26]: the free memory blocks of the heap and of the private pools are represented by a *free list*. When *malloc()* is called, the list is scanned until a block large enough to satisfy the request is found. From this block, a sub-block of the requested size is allocated, and the potential remainder is returned to the free list. This allocation method does not round up the requested memory size to a predetermined size, as is the case with bin-based solutions. It is, as a consequence, more memory-efficient. We implement a *coalescing* strategy: when a memory block is returned (either by *free()* or when a memory block is allocated from a larger block and the remainder is returned), the manager groups it with adjacent blocks in the free list whenever possible. This strategy limits memory fragmentation[26].

In order to keep the number of synchronizations low, a thread calling the *free()* function on a memory block returns it to its own private pool. This operation requires no synchronization, since the considered thread is the only one that can access its own private pool. However, if the *malloc()/free()* ratio is less than 1 for a particular thread, then this thread will tend to accumulate and concentrate the free memory of the system. In turn, this situation could lead to memory starvation of other threads. There are multiple ways to re-distribute the free memory; however, they require

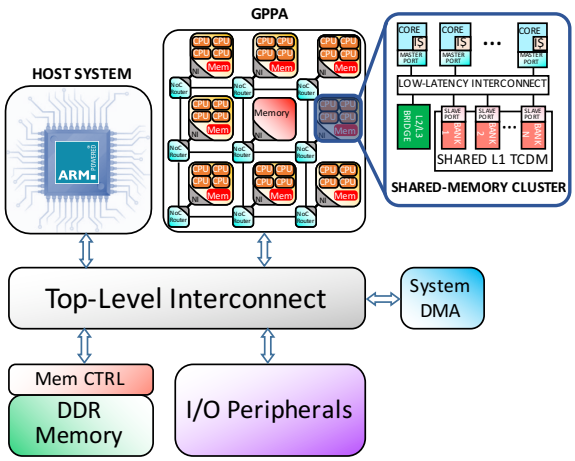



Figure 2: On-chip shared memory cluster

additional synchronizations between threads, and thus potentially downgrade the performance of the application. In practice, we have only encountered applications where the *malloc()/free()* ratio is greater than 1 for all threads, meaning that inside the parallel sections of the considered applications, more memory was allocated than freed (most of the memory is usually freed only at the end of the application run). Consequently this has not been an issue: we consider this particular problem to be out of the scope of this paper, and constitutes by itself a topic of future work.

## 3.2 Evaluation platform

Scaling to large many-core systems is enabled in modern designs by interconnecting tightly-coupled *clusters* with a scalable medium such as a NoC [17] [18]. *Clusters* consist of a small/medium number of simple processing units (PU) sharing high-performance local interconnection and memory, as shown in Figure 2. In our case, each cluster contains 16 RISC32 PUs, each featuring a private instruction cache. Processors communicate through a multi-banked, multi-ported Tightly-Coupled Data Memory (TCDM). This shared level-1 TCDM is implemented as explicitly managed SRAM banks (i.e., scratchpad memory), to which processors are interconnected through a low-latency, high-bandwidth mesh of trees (MoT) data interconnect enabling 1-cycle L1 accesses. As is common in constrained embedded designs, data caches and associated coherency protocols are entirely absent (replaced by the TCDM), allowing for a more scalable and lightweight design. The interconnection supports up to 16 concurrent memory accesses targeting different banks (one port per bank). Since the L1 TCDM has limited capacity (256KB) it is impossible to permanently host all data therein or to host large data chunks. The software must thus explicitly orchestrate data transfers from main memory to L1, for better locality. For performance- and energy- efficient transfers, the cluster is equipped with a DMA engine.

This architectural template captures the key traits of existing cluster-based many-cores such as STMicroelectronics STHORM [18], Kalray MPPA [17], TI Keystone II [2], Adapteva Parallela [1] or the Toshiba Energy Efficient Many-Core [3] in terms of core organization, number of clusters, interconnection system and memory hierarchy. For our evaluation platform, we built a cycle-accurate SystemC simulator, based on the *VirtualSoC* platform [7]. *VirtualSoC* is a prototyping framework. More specifically, we consider a single cluster, plus a memory controller to the off-chip main

memory. To build a many-core platform, clusters can be replicated and interconnected via a Network-on-Chip (NoC) as shown in Figure 2.

The system leverages a partitioned global address space. Each processor in the system can explicitly address every memory segment: local TCDM, remote TCDMs (when multiple clusters are considered) and main memory. The heap spans all the memory banks of the TCDM. Likewise, the private pools that are dynamically allocated from the heap can physically reside in any of these banks.

Processors can synchronize by means of standard read and write operations within a TCDM address range. This provides *test-and-set* semantics, on top of which we implement standard locking mechanisms. In addition, as a synchronization alternative, the described *clusters* are augmented with hardware transactional memory (HTM) support. We leverage the design from Papagiannopoulou *et al.* [22], which targets the same class of shared memory clusters that we consider in this work. This HTM system relies on distributed conflict detection and resolution logic (employing an *eager* scheme), implemented as an extension to TCDM banks, which enables scalable and fast transaction management, as presented in [21].

## 3.3 Transparent refill of private pools

In order to make it robust and more flexible, we want our memory manager to handle the refill of empty private pools in a way that remains transparent to the programmer and the user. A thread running out of memory can request a new, fresh private pool from the shared heap. This process requires extra synchronizations which we implement using transactions. This mechanism must be able to refill a private pool regardless of whether the *malloc()* call that triggered it was performed inside a transaction or not. Next, we present how both cases are treated.

### 3.3.1 *Malloc() called outside of transactions*

Whenever a *malloc()* call placed outside of a transaction cannot be served because there is not enough memory remaining in the corresponding private pool, a *fallback routine* is executed: first the potentially remaining memory in the private pool is returned to the heap, then a fresh private pool is allocated from the heap. After that, *malloc()* is called recursively but will this time allocate from the new private pool. The size of this new heap is computed using the memory size requested in the *malloc()* call and the current private pool size. Since the call was performed outside a transaction, refilling a private pool with just the size requested by the *malloc()* call is enough to guarantee its success (it is, in fact the most memory conservative solution), and in turn to guarantee progress in the execution. However, doing this means that the new private pool is empty again after the call.

If subsequent calls to *malloc()* are performed, the *fallback routine* will have to be executed again, degrading the performance of the application. We chose a more pessimistic (and thus less conservative) tradeoff in our implementation: the refreshed pool has size *max(current_size, double_request)*, where *current_size* is the size of the current private pool when it was allocated, and *double_request* is twice the size requested by the *malloc()* call. This method increases the size of the private pool only when a single *malloc()* by itself requests more memory than half of the private pool, and just refills the pool in other cases. This tradeoff remains memory-

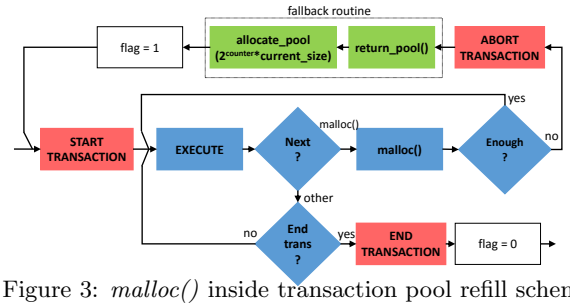



Figure 3: *malloc()* inside transaction pool refill scheme.

conservative while avoiding the need for a large number of pool refills. Indeed, in most cases a pool running out is just the symptom of a *malloc()/free()* ratio greater than 1, and simply refilling the private pool is enough to allow the thread to go on. Each step of the *fallback routine* (which returns the private pool to the heap and allocates a new one) is executed in a dedicated transaction. This results in a total of two transactions in order to synchronize accesses to the shared heap.

### 3.3.2 *Malloc() called inside a transaction*

Calling *malloc()* from inside a transaction is a bit more complicated. Here, having dedicated transactions for the *fallback routine*, as discussed in Section 3.3.1, would result in nested transactions. Since nested transactions are not supported by certain transactional memory systems (and semantics vary for those that do [25]), we chose to avoid them.

A naïve solution would be to execute the *fallback routine* functions without their dedicated transactions. Instead, the initial transaction in which *malloc()* is called would also synchronize the operations of the routine. However, proceeding that way would increase the size and duration of the initial transaction, increasing the probability of aborts.

Thrifty-malloc employs another strategy which has two main advantages compared to this naïve solution. First, it keeps the size and duration of the initial transaction unchanged. Second, it enforces a clear logical separation between the initial transaction and the transactions of the *fallback routine*.

Our strategy is illustrated in Figure 3. If a call to *malloc()* cannot be serviced because a private pool does not have enough memory remaining, the transaction in which *malloc()* was called is aborted. Following the traditional transactional memory behavior, the modifications made to the memory and the processor registers as part of the aborted transaction are reverted. At this point the memory and registers are in the clean state in which they were when entering the transaction for the first time. Then, instead of restarting the transaction right away, as in a normal abort, a *fallback routine* is executed: the remaining memory in the private pool is returned to the heap, and a fresh pool is allocated. Since these operations are performed outside of the initial transaction, their execution is protected by dedicated transactions (as in the case of Section 3.3.1). A flag is raised as a reminder that the pool has already been refilled: should the *fallback routine* be executed again with the flag raised, the size of the private pool would be doubled. The initial transaction is then executed from the start. The flag is cleared only when the transaction commits. The *fallback routine* executed when *malloc()* runs out of memory inside a transaction is thus different than the one described in the previous

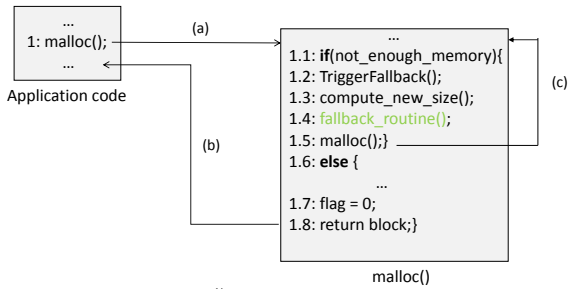


Figure 4: The *malloc()* fallback flow outside a transaction.

section. Indeed, we cannot know *a priori* the cumulated size that will be requested as part of a given transaction, since multiple *malloc()* calls may happen. In order to keep the refill mechanism simple, generic and memory conservative, we use the iterative procedure that we just described.

In the best case, the naïve solution will execute the *fallback routine*, then resume execution and commit without a conflict and thus without having to abort. Our solution, on the other hand, imposes one abort each time a pool is refilled, in the best case. However, in the average case, the probability that additional aborts will occur is much reduced with our solution because it keeps each transaction as small as possible, where the naïve solution enlarges the initial transaction. Moreover, with the naïve solution, if the transaction aborts after the execution of the *fallback routine*, the modifications made to the private pool are reverted during the abort. As a consequence, the *fallback routine* has to be executed again in the next attempt of the transaction run. With our solution, since dedicated transactions are used for the *fallback routine*, the modifications made to the private pool are saved prior to restarting the initial transaction. Consequently, we do not have to run the *fallback routine* again if the initial transaction aborts. To sum up, our solution reduces the probability that aborts occur thereby reducing the overall performance penalty for running out of memory.

## 3.4 Implementation in our manager and simulation platform

Next, we provide a detailed description of the implementation of our memory manager for embedded systems, along with the associated transactional mechanism extensions.

As stated in Section 3.3.2, our method requires support from the HTM. We implement this by providing a primitive called *TriggerFallback()* which is used exclusively in the *malloc()* function. When *malloc()* detects that it cannot find enough contiguous memory in the corresponding private pool, it calls *TriggerFallback()*, which signals the hardware that the *fallback routine* will have to be executed. As previously, we distinguish between two cases, depending on where the *TriggerFallback()* function was called.

If *TriggerFallback()* is called outside of transactions (and thus *malloc()* was also called outside of transactions), the hardware discards the signal, and the execution resumes seemlessly. Indeed, since the call happened outside of a transaction, no hardware support for abort is required, and the algorithm of Section 3.3.1 can be implemented entirely in software. It is done by calling the *fallback routine*, then recursively calling *malloc()*. This sequence of calls is declared inside the *malloc()* function, just after the *TriggerFallback()* call. Figure 4 describes how this is done. When the application calls *malloc()* **(a)**, the function is executed until it reaches a test **(1.1)**: if there is enough memory to allocate a block of the requested size **(1.6)**, the memory block is returned **(b)**, and the application resumes its execution. Otherwise, *TriggerFallback()* is called but ignored by the hardware **(1.2)**, the new size of the pool is computed using the current size of the pool and the size requested by the *malloc()* call **(1.3)**, and the *fallback routine* is executed **(1.4)**. When it is done, *malloc()* is called recursively **(c)**.

If *TriggerFallback()* is called inside a transaction, the corresponding core signals the transactional memory system that it must revert the modifications made as part of the ongoing transaction. Once this is done, the core restores its registers to their values before the transaction started. At this point, the *fallback routine* must be executed before restarting the transaction. This can be implemented in different ways, including:

- Setting the program counter (PC) to the address of the *fallback routine* in memory. Once the *fallback routine* has finished executing, the PC must be set again, this time to the start of the transaction. This solution involves resetting the PC twice and being able to detect the end of the *fallback routine*. Since the *fallback routine* is composed of transactions, the address of the initial transaction must be saved to a special place before setting the PC to the *fallback routine*. Moreover, the programmer must be able to communicate the address of the *fallback routine* to the hardware, since for arbitrary applications and architectures the address may not be the same. This adds complexity to the hardware support.
- Calling the *fallback routine* just after *TriggerFallback()* (as in the non-transactional solution), and branching the PC to the start of the transaction afterwards. Although simpler, this solution still involves saving the address of the transaction to a special place before the *fallback routine* is executed.

In order to keep our solution as simple and general as possible, we choose instead to build a special software primitive on top of the START_TRANSACTION() one, which we call *special_transaction()* in Figure 5. This new primitive wraps a sequence composed of a conditional branch to the *fallback routine*, followed by an unconditional call to the START_TRANSACTION() primitive. This primitive signals the hardware that the execution is entering a transactional section (whose end is marked by a similar END_TRANSACTION() primitive).

By using *special_transaction()* as an *inline* function, a call to the *fallback routine* is guaranteed to be located at a fixed offset before the start of any transaction, in any application. When *TriggerFallback()* is called (and after the memory and registers have been reverted), the core sets the PC to the saved value of the start of the transaction, minus a fixed offset. The control flow is thus redirected to the *fallback routine* call located just before the start of the transaction. When the fallback completes, the PC does not need to be set again, since the start of the transaction is located just after. To avoid executing the *fallback routine* when the execution reaches the transaction for the first time, the *fallback routine* call is guarded by an *if()* construct whose condition always evaluates to false. The only way for the control to reach the *fallback routine* inside the conditional branch is by executing the *TriggerFallback()* statement. We illustrate this implementation in Figure 5.

When the execution reaches the *special_transaction()* inline function, it first evaluates a false condition **(1.1)**, and

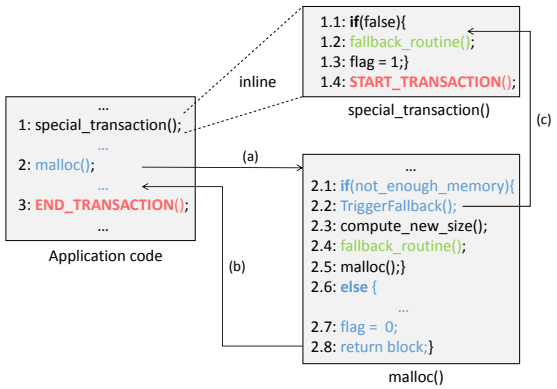



Figure 5: The *malloc()* fallback flow inside a transaction.

jumps directly to the START_TRANSACTION() statement **(1.4)**. Then the code of the transaction is executed, until a *malloc()* call is reached **(2)**. As in the previous case, the function is executed **(a)** and reaches the point where it tests if a large enough block exists in the private pool **(2.1)**. If it is the case **(2.6)**, the iteration flag is reset to 0, the carved block is returned **(b)** and the application resumes its execution until either a memory conflict is detected (leading to an abort of the transaction), or the end of the transaction **(3)** which commits the last allocations, and the flag value reset to 0. If there is not enough memory to serve the request **(2.1)**, the *TriggerFallback()* function is called **(2.2)**. The hardware detects the signal, orders the memory to revert the transactional changes, and resets the registers to their value before the transaction. Then, it modifies the PC **(c)** to redirect the control flow to the *fallback routine* call of **(1.2)**. In this case, the instructions **(2.3)**, **(2.4)** and **(2.5)** are not reached. The execution resumes from **(1.2)**: the *fallback routine* is executed, the iteration flag is set to 1, and the transaction is started again.

We end this section by pointing out that, although we have presented the complete solution as a HW/SW codesign, our software solution is by itself compatible with existing processors featuring HTM. Indeed the Intel Haswell family of processors featuring HTM includes an XABORT instruction that allows transactions to be aborted and a specified fallback software routine is then executed [16]; it is a possible implementation of our *TriggerFallback()* mechanism.

# 4. RESULTS

We evaluate Thrifty-malloc first on a synthetic benchmark to better understand memory pressure behavior, and then on three applications developed as part of the STAMP[20] and MiBench[12] benchmark suites: *vacation*, *genome*, and *patricia*. We chose these three standard benchmarks because they feature dynamic memory allocation inside their critical sections and are well known within the transactional memory community.

To perform the evaluation, we measure the timing performance and memory usage of these applications implemented with Thrifty-malloc and compare them to results obtained with two other memory management strategies. In the first alternative strategy, the applications perform their calls to *malloc()* and *free()* outside of the transactions. This strategy can be adopted whenever a non HTM-compliant memory manager is used in conjunction with transactional memory. It requires extra work from the programmers to take the *malloc()* and *free()* calls out of the transactions, and thus potentially increases the likelihood of errors.

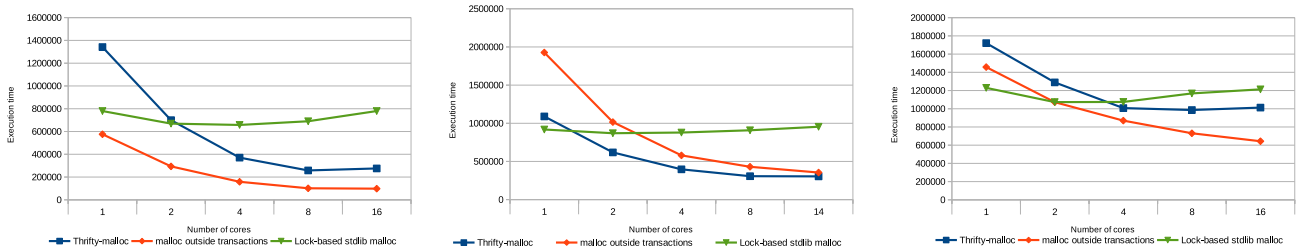
The second alternative strategy consists of synchronizing the applications with locks instead of transactions, and uses a variant of the C *stdlib* memory manager that is optimized for the targeted multicore (simulated) architecture. This strategy is the current *de facto* standard for embedded applications that require dynamic memory management. We emphasize here again the fact that no existing memory manager was able to tackle directly the problem of managing memory inside transactions, and without OS support. As a result, rather than comparing *Thrifty-malloc* with existing managers (which was not possible in our context), we found it more interesting to validate our approach by comparing different memory management *strategies* using *malloc()* implementations that are close to each other in terms of their internal mechanisms (representation of the free memory, allocation routines, etc.), but differ in their synchronization mechanisms, and thus in the way they can be deployed in an application. We show that real differences exist in terms of timing performance and memory conservancy, and observe that our strategy offers the best tradeoff between these criteria while providing the largest opportunity for tunability.

## 4.1 Synthetic benchmark evaluation

As a first experiment, in order to demonstrate the benefits of calling *malloc()* inside transactions instead of only outside, we designed a simple synthetic benchmark composed of a parallelized loop working on a linked-list. Each iteration $i$ starts by a first transaction in which a condition (a shared Boolean variable) is evaluated: if the condition is true, then the executing thread advances to the $i^{th}$ element of the list, removes it from the list and frees it, before leaving the transaction. If the condition is false, the thread leaves the transaction right away. In both cases, the thread then enters a second transaction in which the same condition is evaluated again. If it is true, the thread calls *malloc()*, and adds the newly allocated block at the end of the linked-list. Once again, if the condition is false, the thread just leaves the transaction.

We measure the performance of Thrifty-malloc in terms of timing and pressure put on the heap in this setting, and we compare it to the strategy where *malloc()* can only be called outside of transactions. In this second strategy, *malloc()* is called between the two transactions, regardless of the value of the condition (since the condition is a shared variable in this example, it cannot be evaluated safely outside of transactions). The pressure we measure is the quantity of memory that is currently not available because it has been allocated and was not freed yet, at any given time during the execution. We believe this measure is more accurate than the total quantity of allocated memory considering that we are evaluating a dynamic manager. To clarify, when calculating pressure for the Thrifty-malloc case, we did not consider the private pools themselves since this memory is still available for serving malloc requests by the thread owning the pool (without explicitly freeing it). This is different for the "*malloc()* outside" case since memory may be allocated but never used by the thread, and cannot be used for other malloc requests.

In our experiment, we use the shared Boolean variable to vary the quantity of work that must be performed by the threads as part of the transactions. We consider 4 cases: the condition is true in either 100%, 50%, 30% or 0% of the




(a) Execution time of Vacation (b) Execution time of Genome (c) Execution time of Patricia

Figure 6: Execution time and memory requirements of the benchmarks without pool refill

| | Percentage of working iterations | | | |
|---|---|---|---|---|
| | 0% | 30% | 50% | 100% |
| Thrifty-malloc | 0% | 0% | 0% | 0% |
| malloc outside | 81% | 57% | 40% | 0% |

Table 1: Extra pressure on heap when calling *malloc()* outside of transactions compared to calling it inside

| | 1 core | 2 cores | 4 cores | 8 cores | 16 cores |
|---|---|---|---|---|---|
| Thrifty-malloc | 0 | 6% | 12% | 27% | 60% |
| malloc outside | 0 | 6% | 12% | 19% | 29% |

Table 2: Transaction abort rates for Patricia

iterations.

We display the results in Table 1. As expected, Thrifty-malloc never puts pressure on the heap. Indeed, each *malloc()* call is performed after a corresponding *free()* call. On the other hand, when *malloc()* is called outside of the transactions, since it is not guarded by the execution condition, the extra pressure on the heap — which corresponds to memory leaks or useless, semi-static allocations — grows linearly with the proportion of iterations where the condition is false. It reaches its maximum in the case where the condition is always false (i.e., the 0% column in the table), since *malloc()* is called for nothing at each iteration, and no *free()* operation is performed. In this particular context, the percentages here correspond to the quantity of "leaked" memory compared to the size of the linked list. Each of these percentages also grows linearly with the number of iterations performed.

An interesting consequence for this synthetic benchmark is that for any setting where the condition is not always true, calling *malloc()* outside of the transactions will eventually empty the heap and crash the benchmark: this situation can happen every time a *malloc()* call is guarded by a shared condition in an application. Thrifty-malloc, on the other hand, is immune to this problem and is thus better suited for such applications.

With regard to run time execution for this synthetic benchmark, we realized that placing the *malloc()* inside or outside the transaction has the side effect of significantly altering its data conflict behavior. As a consequence, its execution time is more sensitive to the difference in data conflict abort rates than to the fact that calls to *malloc()* are performed inside or outside of transactions, and therefore we do not report the execution times here.

## 4.2 Performance without pool refill

Next, we present in Figure 6 the results for the STAMP and MiBench benchmarks when no private pool needs to be refilled. In order to ensure that property, we provide each thread with a large enough private memory pool at system startup. In our experiments, using a pool of size 1024 bytes for each but one thread was enough to achieve this. One thread (the main thread) requires (potentially a lot) more memory since it is responsible for initializing some shared data structures, both for the application itself and for the OpenMP management structures, prior to the execution of the parallel section of the application. The execution times we provide here are measurements for the parallel section of each of the benchmarks: we do not take into account the initialization phase where the initial shared structures of the application are being allocated and initialized. Indeed, we focus on the performance of our manager during the parallel phase of the execution, which is common practice when evaluating HTM (or HTM-related) designs.

The first observation we can make is that using hardware transactional memory to synchronize the application scales better than using locks. For both *Vacation* (Fig. 6(a)) and *Patricia* (Fig. 6(c)), the best results are obtained when the memory management is performed outside of the transactions. For these two benchmarks, this strategy is implemented by making all the *malloc()* calls during the initialization phase, and by stocking pointers to the allocated memory blocks inside arrays. This has two important consequences on our measurements: first, the time spent for the allocation (the multiple executions of *malloc()*) is not accounted for since we do not measure the execution time of the initialization phase. Second, since the *malloc()* calls are performed outside the transactions, the duration of the transaction is reduced compared to executing the *malloc()* calls inside. In turn, this reduces the abort rate of the transactions. This phenomenon is particularly present in *Patricia*: in Table 2, we display the abort rates of transactions for both strategies. The abort rate remains limited when *malloc()* calls are performed outside of transactions, but reaches 60% with Thrifty-malloc. These two elements explain why calling *malloc()* outside transactions obtains the best results in *Vacation* and *Patricia*.

For the *Genome* implementation, we decided to implement the "*malloc()* outside" strategy in another way: instead of allocating the memory in the initialization phase, we call *malloc()* inside the parallel section, just before starting the transactions. This way, the execution time of the *malloc()* calls is accounted for in our measurements. As shown in Fig. 6(b), for this benchmark, the two HTM-based strategies deliver equivalent performance for 14 cores, and overall Thrifty-malloc has the best results as soon as parallelism is involved. The reason why the "*malloc()* outside" strategy does not perform better than Thrifty-malloc lies in the fact that it has to overallocate memory. Indeed some transac-

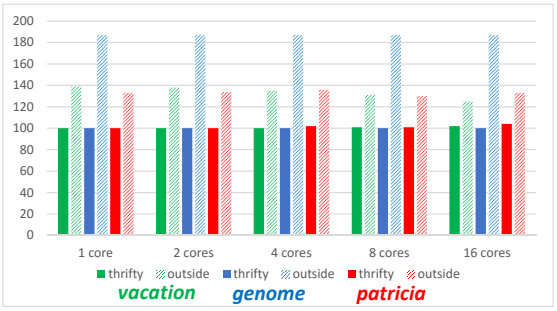



Figure 7: Pressure on heap relative to standard C *malloc()*. *Thrifty* (with solid bars) refers to our method allowing *malloc()* inside a transaction, while *outside* (with dashed bars) refers to *malloc()* outside the transaction.

tions feature conditional branches whose condition can only be evaluated inside the transaction. Since the "*malloc()* outside" strategy imposes calling *malloc()* before the transactions, such conditions cannot be evaluated and the *malloc()* call must be performed in any case. The execution of these additional *malloc()* calls increases the execution time of the parallel section of the application. This overallocation phenomenon is also present in the other benchmarks, but since the *malloc()* calls are performed before the parallel section, we did not measure its influence on performance.

To evaluate the memory usage of Thrifty-malloc, we measure the maximum pressure put on the heap during the execution. Results are shown in Figure 7.

In terms of memory usage, Thrifty-malloc is almost equivalent to the standard C *malloc()* on the three benchmarks. The "*malloc()* outside" strategy overallocates memory for the reason that we explained before. Moreover, for the same reason this strategy cannot perform *free()* calls easily, and the overallocation turns into "over-pressure" on the heap. In *Vacation*, this maximum requirement of memory varies between 39% more than Thrifty-malloc for the single core experiment and 23% more for the 16 core experiment. In *Genome*, the maximum memory required is constant and approximately 87% more than in Thrifty-malloc. In *Patricia*, the maximum memory required varies around 30% more than the pressure generated by Thrifty-malloc.

These results show that Thrifty-malloc handles memory as conservatively as the C standard *malloc()*, while enabling the use of hardware transactional memory and thus with improved performance scaling. We emphasize again that, for embedded systems, total memory capacity is typically severely limited (e.g. the L1 scratchpad memory of the entire cluster is only 256K); therefore, our allocation policy could significantly extend the performance and applicability of the system.

## 4.3 Local pool refill overhead

In Table 3 we measure the cost of refilling a private pool. To do so we decrease the initial size of the private pool, so the threads run out of memory and request a refill at some point during the execution. We compare the measurements with the execution time using Thrifty-malloc when no pool refill is needed (standard C *malloc()* results are also given as an additional reference).

For *Vacation* and *Genome*, the overhead for refilling the private pool is negligible compared to the execution time using locks and the C *malloc()*. We display the number

| | Initial pool size of last core | | | |
|---|---|---|---|---|
| | 1024 bytes | 512 bytes | 256 bytes | 128 bytes |
| **Vacation** | | | | |
| Thrifty-malloc | 100% | 100% | 101% | 109% |
| C malloc() | 265% | 265% | 265% | 265% |
| **Genome** | | | | |
| Thrifty-malloc | 100% | 100% | 100% | 103% |
| C malloc() | 281% | 281% | 281% | 281% |
| **Patricia** | | | | |
| Thrifty-malloc | 100% | 99% | 112% | 109% |
| C malloc() | 120% | 120% | 120% | 120% |

Table 3: Execution time with pool refills relative to Thrifty-malloc without refill (standard C *malloc()* for reference)

| | Initial pool size of last core | | | |
|---|---|---|---|---|
| | 1024 bytes | 512 bytes | 256 bytes | 128 bytes |
| Vacation | 0 | 1 | 2 | 5 |
| Genome | 0 | 0 | 0 | 2 |
| Patricia | 0 | 1 | 1 | 7 |

Table 4: Pool refills as a function of initial size of last pool.

of pool refills executed for each benchmark and each initial pool size in Table 4. For *Vacation*, 1 pool refill is executed when the initial pool size is 512 bytes, 2 when the initial pool size is 256 bytes, and 5 when the initial pool size is 128 bytes. For *Genome*, no refills are necessary for private pools of sizes 1024, 512 or 256 bytes. However 2 refills are executed for an initial pool size of 128 bytes. This jump from no refill to 2 refills can be explained by multiple factors. First, when the first refill happens the private pool may not be empty. The remaining memory may be either non-contiguous, or there may not be enough memory remaining to serve the request. In this case, the non-empty remainder of the private pool is returned to the shared heap during the refill, and the thread will not be able to use it to serve further requests. The second factor is the relative timing of the threads during the execution. A refilling thread has to execute the fallback routine, which modifies its timing relatively to the other threads, compared to an execution where no refill is performed. This adds dynamicity to the system, and modifies the quantity of work each thread has to perform as well as the abort rate of transactions.

This second factor particularly affects the *Patricia* benchmark. No pool refill is needed for an initial size of 1024 bytes. For 512 and 256 bytes, 1 pool refill is executed. However, these refill scenarios occur at different instances of the overall execution, and thus have different consequences on the overall execution time. In the 512 bytes example, the overall execution time is reduced compared to the no-refill experiment. In the 256 bytes experiment, the execution time of the application is increased by around 12%. Finally in the 128 bytes experiment, 7 pool refills are executed, and the execution time is comparable to the 256 bytes experiment. Even though a slight timing overhead occurs when the initial private pool size is 256 bytes or 128 bytes, Thrifty-malloc always delivers a better performance than the standard C *malloc()*.

The pool refill timing overhead in our solution is always negligeable compared to the performances obtained with the standard C *malloc()* combined with locks. Indeed, even in the worst cases where the pool must be refilled seven times our solution still performs better. In the best case, our solution is close to three times faster than the lock-based one. Such a difference is not due to significant differences in the implementation of the allocators themselves, but rather in the fact that Thrifty-malloc enables the system-level use of

HTM for synchronization. The speculative execution in turn allows good performance scaling when increasing the number of cores, while lock-based synchronization does not. As a consequence, we strongly advocate the use of HTM-friendly memory managers such as Thrifty-malloc in the context of multicore architectures featuring HTM.

# 5. CONCLUSION

In this paper we presented a novel method for dynamic memory management for embedded applications and in the presence of transactional memory. This method combines simplicity, flexibility and a lightweight design (in terms of memory usage), making it appealing for embedded systems. We demonstrated its performance on transactional memory benchmarks, and showed that the low overhead incurred by the memory reprovision technique is an acceptable tradeoff for the improved memory usage and the flexibility Thrifty-malloc provides to the system. We also showed that by allowing the use of HTM synchronization, and the management of memory directly inside transactions, our solution allowed a more than 2 times execution speedup on certain applications, and could also reduce the pressure on the heap by nearly half, compared to allocating memory outside of transactions.

In the future, we intend to pursue our work in multiple directions:

- Develop techniques that help maintain a fair repartition of the free memory among the private pools.
- Experiment with more elaborate reprovision patterns.
- Evaluate our memory manager with different transaction abort policies, since abort policies can have a non-negligeable impact on performance depending on the application.
- Extend our memory management method for applications executed across multiple computing clusters.
- Develop a transaction-friendly memory virtualization technique based on a software cache implemented in the transactional memory space, and using DMA calls to communicate with the L3 memory.

# 6. ACKNOWLEDGMENTS

This work was supported in part by NSF grants CNS-1319095, CNS-1519576 and CNS-1301924.

# 7. REFERENCES

[1] Adapteva parallela. http://www.adapteva.com/epiphany-multicore-intellectual-property/.

[2] TI Keystone II. http://www.ti.com/lit/ds/symlink/66ak2h12.pdf.

[3] Toshiba energy efficient many-core. http://www.aspdac.com/aspdac2014/technical_program/pdf/4A-4.pdf.

[4] Y. Afek, D. Dice, and A. Morrison. Cache index-aware memory allocation. In *ISMM*, 2011.

[5] A. Baldassin, E. Borin, and G. Araujo. Performance implications of dynamic memory allocators on transactional memory systems. In *PPoPP*, 2015.

[6] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *SIGOPS Oper. Syst. Rev.*, Dec. 2000.

[7] D. Bortolotti, C. Pinto, A. Marongiu, M. Ruggiero, and L. Benini. Virtualsoc: A full-system simulation environment for massively parallel heterogeneous system-on-chip. In *ISPDP*, 2013.

[8] D. Dice and A. Garthwaite. Mostly lock-free malloc. *SIGPLAN Not.*, June 2002.

[9] D. Dice, T. Harris, A. Kogan, and Y. Lev. The influence of malloc placement on TSX hardware transactional memory. https://blogs.oracle.com/dave/entry/the_influence_of_malloc_placement.

[10] D. Dice, Y. Lev, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Simplifying concurrent algorithms by exploiting hardware transactional memory. In *SPAA*, 2010.

[11] S. Ghemawat and P. Menage. TCMalloc: Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *WWC-4*, 2001.

[13] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[14] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[15] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. McRT-Malloc: A scalable transactional memory allocator. In *ISMM*, 2006.

[16] Intel Corporation. Intel architecture instruction set extensions programming reference manual. Retrieved from http://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf.

[17] Kalray. MPPA 256. www.kalray.eu/products/mppa-manycore/mppa-256/.

[18] D. Melpignano, L. Benini, and E. Flamand. Platform 2012, a many-core computing accelerator for embedded SoCs: performance evaluation of visual analytics applications. In *DAC*, 2012.

[19] M. M. Michael. Scalable lock-free dynamic memory allocation. *SIGPLAN Not.*, June 2004.

[20] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IISWC*, Sept 2008.

[21] D. Papagiannopoulou, A. Marongiu, T. Moreshet, L. Benini, M. Herlihy, and I. Bahar. Playing with fire: Transactional memory revisited for error-resilient and energy-efficient mpsoc execution. In *GLSVLSI*, 2015.

[22] D. Papagiannopoulou, T. Moreshet, A. Marongiu, L. Benini, M. Herlihy, and R. Iris Bahar. Speculative synchronization for coherence-free embedded numa architectures. In *SAMOS 2014*, July 2014.

[23] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, 1995.

[24] O. Shivers, J. W. Clark, and R. McGrath. Atomic heap transactions and fine-grain interrupts. In *ICFP*, 1999.

[25] A. Turcu and B. Ravindran. On open nesting in distributed transactional memory. In *SYSTOR*, 2012.

[26] P. Wilson, M. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Memory Management*. Springer, 1995.