A transaction-friendly dynamic memory manager for embedded multicore systems

Thomas Carle, Brown University, thomas_carle@brown.edu
Dimitra Papagiannopoulou, Brown University, dimitra_papagiannopoulou@brown.edu
Iris Bahar, Brown University, iris_bahar@brown.edu
Maurice Herlihy, Brown University, mph@cs.brown.edu
Tali Moreshet, Boston University, talim@bu.edu

In this paper we present a transaction-friendly dynamic memory manager for high-end embedded multicore systems. The current trend in high-end embedded systems design is to turn to Massively Parallel Processor Arrays (MPPAs) or many-core architectures, and to exploit as much as possible the parallelism they offer. In this context, one of the main performance bottlenecks comes from software synchronization. Although locks are traditionaly used to enforce software synchrony, they suffer from multiple disadvantages such as poor performance and deadlocks. This becomes a big burden for applications that potentially require heavy synchronization on shared objects, such as dynamic memory managers. To alleviate these problems, we propose an original and modular method to perform dynamic memory management using transactional memory. Our memory manager is based on two simple ideas: reduce the number of required synchronizations by provisioning "private" memory for each thread at application startup, and synchronize using transactions instead of locks. Should a thread run out of private memory during execution, the transaction it is running at that time would be aborted, and a mechanism would be triggered to provision more memory to this thread, before reentering the transaction. Using this method, we manage to keep the transactions small, and to clearly separate the transactions that are part of the original application, and those who are dedicated to memory management. Although our method is general and could be applied in any context, we demonstrate its robustness in a particularly demanding environment, namely embedded systems. To do so, we evaluate our memory manager by simulating the execution of benchmarks adapted from classical memory allocator and transactional memory systems publications on an embedded multicore architecture.

1. INTRODUCTION

Existing memory allocators [1; 4] are usually designed for server or database applications, and consequently aim at optimizing performance. This is usually achieved at the cost of an increased memory consumption. In the case of embedded systems, where the available memory space is often limited, this tradeoff may become unacceptable: performance should be optimized while never exceeding certain memory requirements. The use of complex and memory consuming structures is therefore prohibited, and performance must be found elsewhere. Dynamic memory allocators usually rely on the use of a data structure representing the remaining free memory that applications can request from the heap [7]. On a multicore architecture, threads requesting or returning memory have to synchronize their accesses to this structure in order to ensure that it remains coherent. The usual way to implement this synchronization is through the use of locks[4]. Unfortunately, locks are known to be a source of problems ranging from poor performance to difficulty of deployment, especially in complex distributed applications. This difficulty of use is error prone, and can lead to deadlocks which are potentially disastrous for certain embedded applications. An efficient synchronization alternative to locks is to use hardware transactions[3]. In a classical hardware transaction, a thread executes speculatively the code guarded by the transaction until it reaches the end and commits, or until a conflict is found and it aborts. The abort is followed by a revert process that resets the modified memory to the state it was in upon entering the transaction. The execution then restarts from the start of the transaction.

Contribution. We propose a new method for integrating memory management with hardware transactions. In this scheme, a certain amount of contiguous memory (called a local pool) is allocated to each executing thread at application startup. When a thread runs out of memory inside a transaction, the transaction is automatically aborted, followed by a clas-

sical rollback operation. However, the transaction is not restarted right away. Instead, the thread's local pool is reprovisioned with more memory, so that it does not run out again. Once this additional memory has been allocated to the thread, the transaction restarts from the beginning. An important feature is that the memory reprovision mechanism is not executed as part of the same transaction. This method has multiple advantages:

— by keeping the transactions small, and by separating clearly the transactions used by the software, and by the memory reprovision mechanism, it reduces the probability of conflicts, and the data footprint,
— as with classical hardware transactions, this mechanism restores the executing thread to a clean state before allocating additional memory to it and resuming execution,
— it is a very general method which can be applied to any kind of memory allocator (regardless of its optimization criterion), and which works for both hardware and software transactions.

The remainder of this paper is organized as follows: in Section 2, we present in more details the principles of the memory manager that we implemented, and explain the novelties of our method. In Section 3, we present our evaluation method and our results. Section 4 concludes this paper.

2. TRANSACTION-FRIENDLY DYNAMIC MEMORY ALLOCATOR

Our memory manager is based on the sequential fits mechanism[7], where a free list keeps track of all the blocks of memory currently available in the heap. It implements a first fit strategy: the free list is traversed sequentially from its start, and the search stops as soon as a free block of size equal or superior to the requested size is found. A block of the requested size is carved from it and returned to the requesting thread, while the potential remainder is returned to the free list.

In order to reduce the number of synchronizations on this free list, our allocator works with local memory pools. At application startup, each thread[1] requests a certain amount of contiguous memory from the heap to serve as a local pool. Synchronizations on the heap free list are ensured using transactions. After this initialization phase, memory requests by the threads are allocated directly from the local pools, and thus require no extra synchronization. In the same fashion, memory blocks freed by a given thread are returned to its local pool without further synchronization.

2.1. Advanced features

For regular applications where memory requirements are not too high, we expect to be able to predict their needs in a static fashion (or to simply overallocate memory if possible), so that most or all online allocations will be made directly from the pool without further synchronization. Nevertheless, this is not possible for all applications. For applications with a high inherent degree of dynamicity, or when it is not possible to predict in advance how much memory will be needed nor when (for example in applications that depend on user-defined input, or which must react to events), our memory manager must be able to allocate additional memory to the local pools as needed. Although less performant than allocating directly from the local pools (synchronization is needed on the heap free list), this scheme makes the memory manager more robust in dynamic situations, by allowing the application to continue its run instead of crashing.

This feature was implemented by modifying the transactions scheme in order to allow the execution of a fallback routine. When a thread runs out of memory in its local pool (or the remaining memory is too fragmented to be used), the allocator automatically triggers the abort of the currently running transaction, which restores the original clean state the

[1]Our architecture currently supports only one thread per core

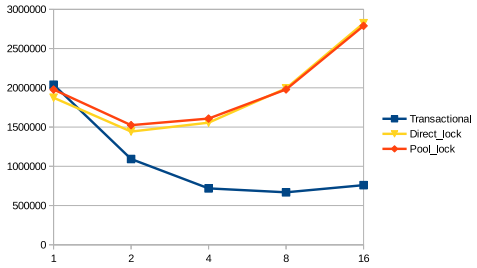

Fig. 1. Execution times for the *vacation* benchmark with various numbers of cores running

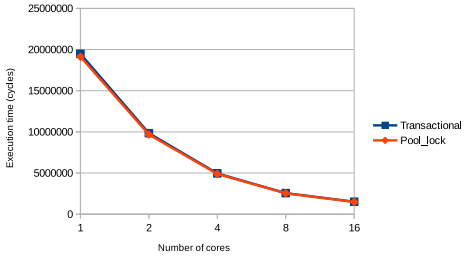

Fig. 2. Execution times for the *threadtest* benchmark with various numbers of cores running

thread was in when entering the transaction (which includes reverting all memory writes by the thread during the aborted transaction). A fallback routine is then executed to refill the local pool with fresh memory from the heap, before reentering the transaction. This fallback routine is not executed as part of the same transaction: it can be synchronized using locks or using a dedicated transaction. In its current version, our memory manager uses two consecutive dedicated transactions: the first deallocates the potentially remaining elements of the local pool, and the second reallocates a fresh (contiguous) pool of larger size (we double the size of the pool).

3. EVALUATION

We first evaluated our memory manager on the *vacation* benchmark from the STAMP suite[5]. We simulated the execution of this benchmark on our platform[6; 2], and measured the running time of the application without considering the initialization phase (nor the equivalent deallocation phase at the end of the run), since they are not relevant for embedded systems. We thus start our measurements when the environment has been initialized, and all cores have requested and obtained their local pools. We compare three ways of allocating memory and synchronizing the application:

— in *Transactional*, we use our memory manager with local memory pools dedicated to the cores, and synchronize using transactions,
— in *Pool_lock*, we use local memory pools, but synchronize using a lock,
— in *Direct_lock*, we do not use local memory pools. Instead, the cores have to request their memory directly from the heap using a lock. In order to synchronize the rest of the application, we use a second lock.

The results displayed in Figure 1 were obtained by running the *vacation* benchmark on simulated architectures composed of respectively 1, 2, 4, 8 and 16 cores. The first observation is that the use of local pools combined with transactional synchronization outperforms by far the lock-based synchronization schemes. Until 8 cores, the performance scales up as expected, while the lock solutions see their performance decrease as soon as more than two cores are running. This performance degradation comes from the increased contention to access the shared structures of the application: locks serialize these accesses and thus reduce the benefits of executing in parallel. Moreover, the increased contention for lock acquisition itself generates non-negligible overhead. The *Transactional* scheme performance stops scaling after 8 running cores, and its performance decreases when going from 8 to 16 cores. We believe this counterperformance is due to the nature of the benchmark: as we pointed out with the lock-based results, the contention becomes very high with 16 cores. We measured that the 8-core run generated 7 conflicts, and the 16-core 18 conflicts. In our setting, this corresponds to adding 14 percent of extra work to the original application (by re-executing the aborted transactions).

We also experimented our allocator on the *threadtest* benchmark described in [1]. In this benchmark, we repeatedly allocate 10000 blocks of size 8 bytes and then free them in order. The results are displayed in Figure 2. As expected, our allocator shows a performance scaling when the number of cores is increased. Indeed, by using local pools we suppress all need of synchronization on the heap free list, and this benchmark does not require any other type of synchronization. Since no synchronization is needed, it has no impact to use transactions or locks as long as large enough local pools have been allocated during the initialization. The results for the direct allocation on the heap using locks are not displayed on the figure, because this scheme performs very poorly (around 2 orders of magnitude more than the pool-based managers).

In order to measure the efficiency of our reprovision scheme, we executed a series of simulations on the *vacation* benchmark, in which we varied the size of the local pool allocated to one of the cores. In this setting, we simulate an execution platform composed of 8 cores. For all but one core, we allocate a local pool with enough memory to execute the complete run. On the last core, we progressively decrease the size of the initial pool in order to trigger the reprovision mechanism. The results are displayed in Figure 3. We measured the simulation time for local pools of initial size 256 bytes, 512 bytes and 1024 bytes. Our first observation is that 1024 bytes were sufficient to run the simulation without requesting additional memory. In both of the other cases, the memory reprovision mechanism was triggered only once. Overall, we observe that the overhead induced by our reprovision scheme is small (approximatively 0.7% of the whole run). The 256 bytes pool runs out of memory early in the simulation, and the corresponding thread is reprovisioned with 512 fresh bytes to finish the run from there (the allocations that were already made do not have to be executed again). The 512 bytes pool runs out near the end of the simulation. Since both pools do not run out of memory at the same point, we expect their configurations to be slightly different when they are freed as part of the reprovision scheme: for example the amount of potentially remaining memory in both pools (yet too small or fragmented to be used) may differ. This explains the difference between the measured overheads in these two simulations.

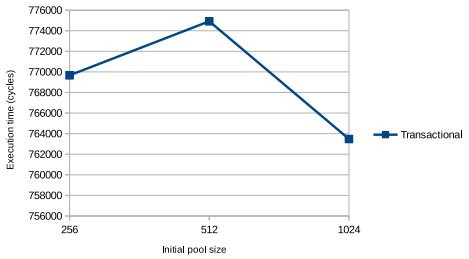

Fig. 3. Execution times for the *vacation* benchmark while varying the size of the local pool of a core

4. CONCLUSION AND PERSPECTIVES

We developed an original method to support dynamic memory management in a transactional context. It is based on the allocation of a local memory pool to each executing thread, and on the modification of a hardware transactional memory scheme in order to reallocate memory upon abort when a local pool is empty. Our initial results show significant performance advantages in using this technique compared to lock-based strategies. As part of future work, we will try our memory manager on more benchmarks to better assess its performance. We will also measure the cost of reprovisioning multiple pools during the execution, and see how it impacts the performance of our method. Another interesting perspective is to further investigate our fallback reprovision mechanism. A first improvement would be to assess whether reallocating a full local pool is necessary, or if allocating a single block of the needed length is more efficient. A more ambitious step will be to allow one thread to steal memory from the local pools of other threads in the case where there is no more fresh memory in the heap. This implies that one thread could impose the abort on other threads running transactions, before reallocating local pools. By carefully implementing this scheme where each thread can be helped by the other threads, we expect to increase predictability and performance when contention and memory requirements are high.

REFERENCES

[1] 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.

[2] D. Bortolotti, C. Pinto, A. Marongiu, M. Ruggiero, and L. Benini. Virtualsoc: A full-system simulation environment for massively parallel heterogeneous system-on-chip. *2013 IEEE International Symposium on Parallel and Distributed Processing*, 0:2182–2187, 2013.

[3] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[4] M. M. Michael. Scalable lock-free dynamic memory allocation. *SIGPLAN Not.*, 39(6):35–46, June 2004.

[5] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46, Sept 2008.

[6] D. Papagiannopoulou, T. Moreshet, A. Marongiu, L. Benini, M. Herlihy, and R. Iris Bahar. Speculative synchronization for coherence-free embedded numa architectures. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, July 2014.

[7] P. Wilson, M. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Memory Management*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1995.