Early Results Using Hardware Transactional Memory for High-Performance Computing Applications*

Karl Ljungkvist
Uppsala university
kalj0193@student.uu.se

Martin Tillenius
Uppsala university
martin.tillenius@it.uu.se

Sverker Holmgren
Uppsala university
sverker.holmgren@it.uu.se

Martin Karlsson
Uppsala university
martin.karlsson@it.uu.se

Elisabeth Larsson
Uppsala university
elisabeth.larsson@it.uu.se

ABSTRACT

Transactional memory (TM) is a novel approach for handling concurrency issues. By experimenting on a prototype system implementing transactional memory in hardware, we aim at investigating if TM is of benefit for scientific and high-performance computing. The main goal of our work is to exploit the optimistic nature of transactional memory and increase performance by reducing the overhead from locks and thread synchronization. Preliminary results are presented for a basic particle dynamics computation, which is one of the most widely used HPC algorithms.

We show that using hardware transactional memory gives a performance improvement over corresponding implementations using locks, which can be as big as a factor of two for the extreme cases.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*

General Terms

Performance, Experimentation, Design

Keywords

Hardware, transactional memory, performance, high performance computing

1. INTRODUCTION

When using the shared memory model for parallel programming, communication is trivial since each thread can access the memory used by all other threads. However, because of the concurrent execution, this also means that race conditions may occur with unpredictable results when several threads compete for the same shared resource. To resolve this, such critical sections must be executed atomically.

The classical way of achieving atomicity is to protect critical sections by mutually exclusive locks which keep the threads from executing critical sections at the same time. However, there are still many problems that can arise in a concurrent code using locks, e.g. deadlocks, convoying and priority inversion [6]. Convoying occurs when a thread holding a lock gets descheduled, e.g. if a page fault occurs, or if its scheduled time runs out. This might lead to a situation where the threads that are about to run the critical section cannot do so until the descheduled thread is rescheduled. Priority inversion occurs when a high-priority thread is hindered from running by a low-priority thread holding a lock. Apart from these problems, there is also the extra cost of acquiring and releasing locks each time a critical section is executed in the code.

Another issue with using a lock-based implementation is that in some applications there is rarely any contention for the locks. This means that the threads will be spending a lot of time acquiring and releasing locks although the locks are only rarely actually needed. Two examples of such operations of quite different origin and type are the enqueue-dequeue operation [6], where both ends of the queue have to be locked, even if only one end is actually accessed, and assembling the stiffness matrix in finite element methods[7] for solving differential equations, where one lock per element in the matrix is needed. In this context, locks can be considered to be pessimistic; even if a lock is hardly ever needed, it must still be acquired to guarantee correct results.

2. TRANSACTIONAL MEMORY

An alternative approach for solving the concurrency issues mentioned above is to use Transactional Memory (TM). Here, a transaction is defined as a section of code which can either commit (succeed) or abort (fail). If a transaction commits, all changes it has made to the memory are made permanent, and if it aborts, all changes are rolled back and the pre-transactional state of the system is recalled. Because of this, the transaction is atomic.

There are different conditions that can make a transaction fail, but in this context the most interesting one is in the

*This work was supported by the Linnaeus center of excellence UPMARC (Uppsala Programming for Multicore Architectures), funded by the Swedish Research Council

event of concurrency. If two transactions read and write the same piece of memory simultaneously, at least one of them will fail. When a transaction fails, the simplest measure is to just retry the transaction as soon as possible, but more advanced schemes have also been presented, for instance exponential back-off.

TM can solve some of the problems associated with locks, e.g. the overhead problem when the contention is low. With TM, instead of locking the critical sections, they are just executed and if a conflict is detected, at least one of the threads must abort its transaction. Therefore using transactions is an optimistic approach. It first tries to run the critical section and then handles problems when they occur (if they do). However, in programs with high contention one might probably as well use locks, since if TM is used, most of the transactions will fail to commit due to conflicts, and the efficiency will drop.

There are several ways to implement transactional memory. One way is through Software Transactional Memory (STM), where a software library is implemented to provide the necessary operations to perform the transactions. Two examples of such libraries are TL2[5] and SkySTM[8]. An STM library will have to monitor the memory accesses in the transactions and abort a transaction in the event of a conflict. It also has to include a fail policy defining what to do in the event of an abortion (simple restart, back off, etc). An obvious problem with STM is the substantial overhead due to the extra instructions added when making calls to the library, as well as the conflict detection code.

There are several ways to implement Hardware Transactional Memory (HTM). Already in 1993, Herlihy and Moss showed that HTM could be implemented as a modified cache coherence protocol, with very successful simulation results [6]. A more thorough investigation of the design choices that exist when implementing an HTM system was done by Bobba et al. [2].

A problem with hardware implementations that is not prominent in software solutions is that the memory available for transactions is typically quite limited. A solution to this was proposed by Ananian et al. in their Unbounded Transactional Memory implementation (UTM), where speculative memory is kept in cache but is allowed to spill out into the main memory if necessary [1].

Our study was performed on a prototype system with support for best-effort transactional memory in hardware. The interface to the TM consists of two instructions, one which starts a transaction, and one which commits it. There is also a register containing information about the reason for a transaction failure. The processor of the system has 16 cores arranged in clusters of four, where each cluster shares an L2 cache of 512 KB (2 MB in total). The system has a total of 128 GB of RAM. For a detailed description of the system, see [3].

3. APPLICATION

A simple particle dynamics simulation in 2D was chosen for our study. This simulation is a typical example of how the movement of mutually interacting "particles" in a general sense (ranging from atoms in molecular dynamics to stars and galaxies in astrophysics) is predicted in a vast range of computational science applications.

In the standard force-field models, including e.g. electromagnetic forces and gravitation, the particles interact pairwise with each other. When the force between a pair of particles has been calculated, both particles need their force variable updated. When the task to calculate the forces is distributed over several threads, two threads must not try to update the force variable of the same particle simultaneously. Such collisions are expected to be fairly rare, making the application suitable for transactions, exploiting their optimistic nature.

We evaluate the interaction between each pair of particles, meaning that we need to evaluate $O(n_p^2)$ interactions in each time step, where n_p denotes the number of particles. In more advanced algorithms, attempts are made to avoid calculating all interactions. There are many different methods for doing so, such as ignoring long-range interactions, exploiting periodicity, or grouping distant particle together. However, at least for particles within a short distance from each other, the interactions need to be evaluated explicitly, and this is the scenario we focus on here.

4. IMPLEMENTATION

A straight-forward serial implementation of the force evaluations looks like this:

Listing 1: Simple serial implementation

```
for i = 0 to n_p - 1
  for j = i + 1 to n_p - 1
    Δf = evalForce(p_i, p_j)
    f_i += Δf
    f_j -= Δf
```

Here f_i is the total force acting upon particle p_i, and Δf is the contribution to f_i from particle p_j.

4.1 Parallel Implementation

All parallel implementations were done using the pthreads library. When parallelizing the algorithm in Listing 1, care must be taken to avoid two threads from concurrently updating the force acting on the same particle. Also, just dividing the outer loop up in even chunks and distributing them over the cores give bad load balancing, as the amount of work in the inner loop varies. For better load balancing, we instead distribute the loop over i in a cyclic manner over the threads. This assignment to threads can either be done statically, as in Listing 2, or dynamically, by making the outer loop index variable global and letting all threads read and increment it atomically.

Listing 2: Parallel implementation

```
for i = id to n_p - 1 step by n_t
  for j = i + 1 to n_p - 1
    Δf = evalForce(p_i, p_j)
    atomic { f_i += Δf }
    atomic { f_j -= Δf }
```

Here id is the thread number and n_t is the number of threads. The `atomic` keyword means that the scoped operation must be executed atomically. This can be implemented using locks, or by performing the operation in a transaction.

4.2 Implementation Using Locks

When using locks, we use pthread mutexes in our implementations. We have also performed tests using the pthread spinlock which gave similar but slightly worse results in all tests and is excluded here for clarity. To avoid false sharing, each lock was allocated its own cache line.

The first implementation, called **basic locks** follows the code in Listing 2, and uses one lock per particle. This leads to a lot of locks and too much time spent on lock handling. To improve this, we can let each lock protect a group of particles, but not too many or too much time will be spent waiting for locks instead. In our case it turned out that grouping particles into groups of 4 gave the best performance.

When dynamic scheduling is used, the contention on the lock protecting the global index can be high. This issue can be addressed by assigning whole blocks of indices to threads, rather than treating them individually. Also here, we found that blocking indices into groups of 4 gave the best performance. The implementation with calculations on blocks of 4 particles, and indices distributed in blocks of 4 is called **blocked locks**.

4.3 Implementation Using CAS

The update can also be made atomic by using the atomic instruction compare-and-swap. Using this method, it is only possible to update a single value atomically, but not for instance the whole 2-dimensional force, as the other methods do. For our application, this does not matter. A hinder is that the compare-and-swap instruction only works on integer registers, while we want to update floating point values. This means that values must be moved back and forth between floating point registers and integer registers. To do this, the values have to be stored and reloaded, which is quite expensive. This implementation is called **basic CAS**, and we also implemented a version grouping particles and indices as in the locks implementation, called **blocked CAS**.

4.4 Implementation Using Private Buffers

A different approach is to trade concurrency for memory and letting each thread have its own private buffers for calculated force contributions to which the thread have exclusive access. These private buffers are then added to the global variables in parallel when all force contributions have been calculated. This requires $O(n_p n_t)$ memory instead of $O(n_p)$, but completely avoids concurrent updates of the forces. It also requires a more substantial modification of the original algorithm than in the other approaches. This implementation is called **private buffers**.

4.5 Implementation Using Transactions

For the implementations using transactions, the forces are updated within transactions to get atomic updates. We use the same grouping strategies as when using locks, to get the same memory access patterns. The implementation that updates a single particle in each transaction, is called **basic HTM**, and the implementation with particles and indices are blocked into groups of 4, is called **blocked HTM**.

An important result of our early experimentations on the test system was that transactions could fail to commit for a number of reasons. We thus have to decide what to do when a transaction fails. Simply retrying the same transaction until success turned out not to work well in our experiments. The program repeatedly got stuck in infinite loops when a transaction never was able to succeed.

In our experiments, 5 different fail codes accounted for over 99 % of the failed transactions. We will call these **coherence:** for transactions failing due to conflicting memory accesses, **load:** for failures due to problems reading memory, **store:** for failures when writing to memory, and **other:** for all other problems (e.g. interrupts, too large transactions, or trying to execute certain unsupported instruction within an transaction). A much more thorough investigation of the error codes and their meanings is given in [4].

The following approach for handling failed transactions resulted in the best performance. If a transaction fails with a **store** error, we use a trick from [4] and perform a dummy compare-and-swap (if the contents was zero, write zero) to the address to initiate an appropriate TLB and cache load, and then retry the transaction. If a transaction fails with a **load** error, we read the data from outside the transaction and loop until the data is ready before retrying[1]. If the transaction fails due to a **coherence** error, we used an exponential back-off strategy where we waited for a random time between each retry, and let the upper limit of this random time increase with the number of retries, up to a certain limit.

In practice, it turned out that the trick with the dummy compare-and-swap operation was needed so frequently that always doing the compare-and-swap when a transaction failed turned out to be the fastest approach.

5. EXPERIMENTS

To study the distribution of run times each implementation was executed 100 times, and the median of the run time is reported here. In each execution, 1024 particles were simulated for 40 time steps. The execution time (wall-clock time) was only measured for the last 20 time steps to concentrate on the behavior of long runs and filter out possible initialization issues such as filling the caches. Storing 1024 particles only required 56 kB of memory in our implementations, meaning that all data easily fit in the L2 cache. In all experiments, each threads is pinned to its own microcore.

6. RESULTS

To see if TM can give better performance for little implementation effort, the straight-forward implementations that closely follows Listing 2 were compared against each other. The results are shown in Table 1 and Figure 1. The dashed line labeled "ideal" starts from the execution time of a basic serial implementation as in Listing 1, and scales perfectly

[1]The loop is performed using the architecture-specific *branch on register ready* instruction to determine when this data is ready.

with the number of threads. The transactional memory version outperformed the basic locks version and was the best when running on fewer than 16 cores, but was beaten by the compare-and-swap version when running on all cores. This agrees with the assumption that transactions are good when the contention is low, but less effective when the contention is higher and many transactions fail and need to be retried.

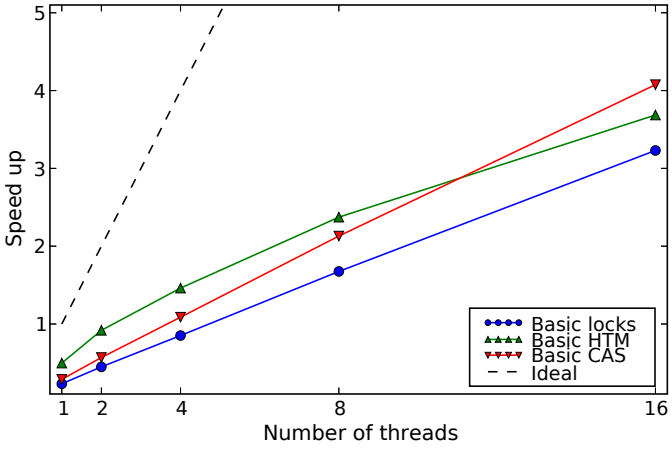

Figure 1: Median speed up relative to serial

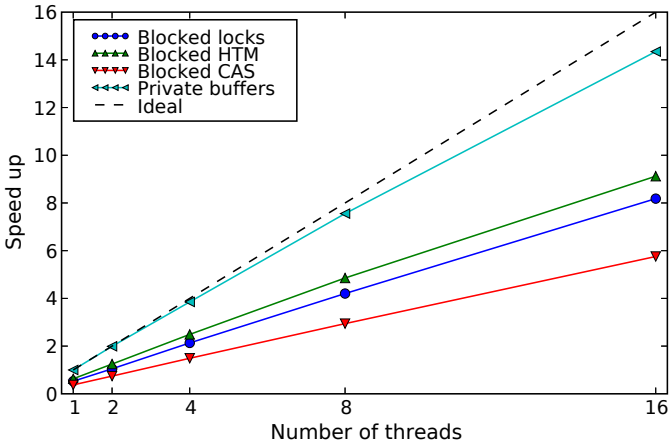

Figure 2: Median speed up relative to serial

Figure 1 visualizes the speed up for the basic implementations. When running on one thread and comparing against the basic serial implementation, the overhead from lock handling causes an over four times longer execution time, and the overhead from transactions causes a doubling of the execution time. We can also see that the implementations using locks and compare-and-swap scale linearly with the number of threads, while the increase in speed up for the version using transactions slightly decays when the number of threads increases.

The results for the implementations that groups particles and indices to reduce overhead are shown in Figure 2. Here the private buffers version running on one thread performed better than our other serial implementations, so the ideal line is here taken to start at the run time of the private buffers implementation, but with perfect scaling. Note that this is different than the ideal line in Figure 2.

Table 1: Median run-times (seconds)

N_{threads}	1	2	4	8	16
Basic serial	2.06	-	-	-	-
Basic locks	8.91	4.59	2.42	1.23	0.64
	(1.0)	(1.9)	(3.7)	(7.2)	(13.9)
Basic HTM	4.14	2.24	1.41	0.87	0.56
	(1.0)	(1.8)	(2.9)	(4.8)	(7.4)
Basic CAS	7.16	3.62	1.89	0.97	0.50
	(1.0)	(2.0)	(3.8)	(7.4)	(14.3)
Blocked locks	2.38	1.19	0.59	0.30	0.15
	(1.0)	(2.0)	(4.0)	(7.9)	(15.9)
Blocked HTM	1.99	1.00	0.50	0.26	0.14
	(1.0)	(2.0)	(4.0)	(7.7)	(14.2)
Blocked CAS	3.35	1.69	0.84	0.43	0.22
	(1.0)	(2.0)	(4.0)	(7.8)	(15.2)
Private buffers	1.25	0.63	0.32	0.17	0.09
	(1.0)	(2.0)	(3.9)	(7.4)	(13.9)

Looking at Figure 2, we see that both the locks version and the CAS version have higher overheads than the version using transactions. The private buffers implementation is the clear winner, and shows that trying to avoid concurrency is the best strategy, when possible. This version also deviates the most from Listing 2, as it includes an additional step where each threads force calculations are merged together. The other implementations all have exactly the same structure, and the only thing that differs is the method for atomically updating the force variables. The compare-and-swap method, that showed much promise in Figure 1, ended up at the bottom this time. A possible explanation of this is that the overhead is higher than for locks or transaction when the contention is low. We also note that the implementation using transactions scales better now when the contention is lower, and it ended up slightly faster than the locks implementation.

7. CONCLUSIONS

Our results from both the naive algorithm and the blocked algorithm comparison show that transactions can give a decent speed up at no extra implementation effort when compared to locks. Using transactions also reduces the memory footprint, as no memory is needed to store any locks.

Using atomic instructions, in this case compare-and-swap, to update floating point variables turned out to be slower than the other alternatives when the contention was moderate. The reason this method did not work out well is the need to move data back and forth between the floating point registers and the integer registers, which caused a lot of overhead.

The best results were achieved by completely avoiding concurrent updates, as expected. This usually requires a significant redesign of the code and is not always possible. In the case of fine-grained synchronization of memory accesses, TM reduces the need to analyze and fully understand the memory access patterns of the algorithm.

During our work we have noted that optimization of the failed-transactions handling has yielded a significant performance gain and we believe there is still much room for improvement there. This could make the advantage for transactions over locks and atomic instructions even larger.

8. REFERENCES

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[2] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34rd Annual International Symposium on Computer Architecture. International Symposium on Computer Architecture*, pages 81–91, 2007.

[3] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A high-performance Sparc CMT processor. *IEEE Micro*, 29:6–16, 2009.

[4] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 157–168, New York, NY, USA, 2009. ACM.

[5] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, 2006.

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

[7] M. G. Larson and F. Bengzon. *The Finite Element Method: Theory, Implementation, and Practice.* Department of Mathematics, Umeå University, 2009.

[8] Y. Lev, V. Luchangco, V. J. Marathe, M. Moir, D. Nussbaum, and M. Olszewski. Anatomy of a scalable software transactional memory. In *2009, 4th ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2009.