6.895 Theory of Parallel Systems Sean Lie

An Integrated Hardware-Software Approach to Transactional Memory

1 Abstract

Transactional memory has been proposed as a general and flexible way to allow programs to read and modify disparate primary-memory locations atomically as a single operation, much as a database transaction can atomically modify many records on disk. Hardware transactional memory supports atomicity through architectural means, whereas software transactional memory supports atomicity through languages, compilers, and libraries. Hardware transactional memory can be implemented quite easily with some modificaitons to the processor cache and cache controllers. However, this implementation imposes limitations on the size and length of transactions. On the other hand, software transactional memory does not suffer from these limitations but has a much higher overhead than hardware transactions.

In this paper, I propose an integrated hardware-software implementation of transactional memory. The integrated approach gives us the best of both hardware and software transactions. It allows small and short transactions to run in hardware with very low overhead. In the common case, transactions are small and short so it is desirable to run them fast. On the other hand, large and long transactions run slower in software but are nevertheless possible and fully supported. Moreover, the high overhead is also amortized over the common case and thus does not have a significant penalty on overall performance.

2 Introduction

In shared-memory parallel programming, often the ability to perform several memory operations atomically is required for correct program execution. Achieving such atomicity in shared memory is a nontrivial task. Traditionally, locking has been used to solve this problem. Unfortunately, locking is not natural for programmers and thus can often lead to undesirable results such as deadlock and priority inversion when used incorrectly.

Even the simple task of locking two independent objects can result in deadlock if done incorrectly. Consider example code given in Figure 1 that pushes flow X from node i to node j. Running this code in a parallel environment will not always give the desired result since another processor may also be running the same code and modifying the same nodes. Therefore, we need to ensure that the push of flow happens as an atomic action.

Figure 2 shows the added complexity associated with using locking to achieve atomicity. Notice that we need to order the lock acquisition to avoid deadlock. Deadlock avoidance is pretty simple in this case, but it is still not trivial. In more complicated situations, reasoning about deadlock becomes much more difficult and often leads to conservative locking. Programmers use conservative locking in the interest of correctness and ease of implementation. However, it can result in very poor performance since it can hide the available parallelism in the code.

```
node_push(i,j) {
  Flow[i] = Flow[i] - X;
  Flow[j] = Flow[j] + X;
}
```

Figure 1: Code to push flow X from node i to node i without any synchronization.

```
node_push(i,j) {
  if (i<j) {
    a = i;
    b = j;
  } else {
    a = j;
    b = i;
  }
  Lock(L[a]);
  Lock(L[b]);
  Flow[i] = Flow[i] - X;
  Flow[j] = Flow[j] + X;
  Unlock(L[b]);
  Unlock(L[a]);
}
```

Figure 2: The `node_push` code using locking for synchronization.

In addition, even simple lock ordering adds a significant overhead. Not only do we need to run more instructions, we are adding an unpredictable branch. Locking incurs a high space overhead as well since locks are simply just memory locations.

Moreover, the very concept of locking is inherently conservative. A lock may be obtained and released without any attempts by other processors to obtain that lock. Such a situation is very common since contention in highly parallel applications is usually low. Therefore, locking is not necessary for the most part, but is nevertheless required to ensure correctness in the few cases where a conflict does occur. Since the action of obtaining and releasing a lock can be very expensive, the use of locks (even when optimized) can lead to suboptimal performance and unnecessarily high resource overhead.

Ideally, the programmer should have the ability to perform several memory operations atomically without having to worry about all the issues associated with locking. Transactional memory achieves this goal. Transactional memory is intended to replace conventional locking techniques to achieve higher performance and a more intuitive programming environment. Figure 3 shows how transactional memory primitives would be used to achieve atomicity for the `node_push` example. The underlying system will ensure that all instructions executed between a `StartTransaction` and an `EndTransaction` will be atomic.

This paper describes an implementation of the underlying system that uses both hardware and software constructs. The remainder of this paper is organized as follows. Section 3 describes the

```
node_push(i,j) {
  StartTransaction;
  Flow[i] = Flow[i] - X;
  Flow[j] = Flow[j] + X;
  EndTransaction;
}
```

Figure 3: The `node_push` code using transactions for synchronization.

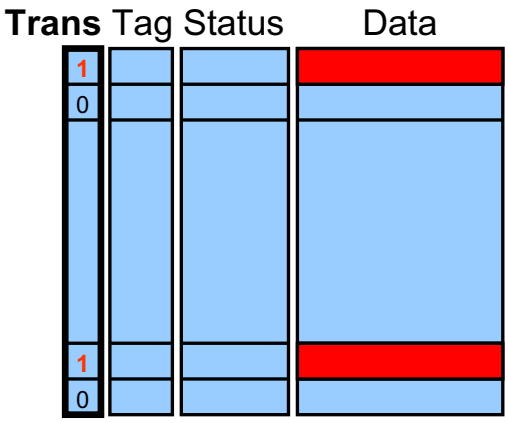

Figure 4: The cache in HTM. 1 bit (Trans) is added per cache line to indicate that the data stored in the line is transactional. Transactional data is shown in red.

hardware transactional memory mechanism and its limitations. Section 4 describes the software transactional memory mechanism. Section 5 argues why an integrated approach is desirable. Section 6 describes the integrated approach and Section 7 evaluates its performance. Section 8 provides some concluding remarks.

3 HTM: Hardware Transactional Memory

HTM is a transactional memory system implemented completely in hardware. HTM is based on the transactional memory system originally designed by Herlihy and Moss [2, 3]. HTM requires only minor changes to the cache and cache controllers. I will use HTM as the baseline hardware transactional memory system.

In HTM, all transactional data is stored in the cache speculatively. As shown in Figure 4, all transactional data in the cache is marked using an additional bit per cache line. Transactional data is not written back to main memory unless the transaction has committed. Such a constraint ensures that main memory always has consistent data while the cache may contain speculative transactional data.

HTM uses the cache coherency protocol to detect conflicts between transactions. If another

processor attempts to access any of the addresses marked transactional, the cache will receive an intervention message instructing it to either change the state of the cache line or discard the cache line altogether. In such a case, there is a conflict and the transaction is aborted.

Transaction abort consists of discarding all data written by the transaction thus ensuring that the memory system appears as it did before the transaction was started. Transaction abort is accomplished by flash clearing all the transactional bits in the cache as well as invalidating any transactional cache lines with modified data.

If the transaction completes successfully without any conflicts, the transaction is committed. Transaction commit is accomplished by simply flash clearing all the transactional bits in the cache. Removing all transactional bits allows transactional data to be written back to main memory thus making all transactional changes globally visible.

HTM has very low overhead. Since all cache actions are done atomically in hardware, transactional memory operations suffer essentially no performance impact compared to normal memory operations. I present some performance overhead results in Section 5. As those numbers show, the overhead is not zero but it is very low when compared to using locking to achieve atomicity.

Since HTM uses the cache to store transactional state, it suffers from two very serious limitations. The first limitation is size. The size of a transaction is the number of memory locations that it can touch. The second limitation is length. The length of a transaction is the number of cycles it takes to complete the transaction.

3.1 Hardware Size Limitation

Since HTM stores all of its transactional state in the cache, the size of the transaction is limited by the size of the cache. HTM cannot execute transactions that do not fit in cache. When a transactional cache line needs to be evicted from the cache, the transaction is aborted. Therefore, HTM aborts all large transactions.

The size of the cache is not the only factor that determines the size limit of a transaction. Cache parameters such as associativity and line size may cause a transactional eviction well before the entire cache is used for transactional data. This also implies that the precise size limit is difficult to predict as it may depend on many factors.

3.2 Hardware Length Limitation

HTM transactions are also limited in length since HTM aborts the running transaction when a context switch occurs. This is necessary since the transactional state (the cache) is linked to the processor and not a thread. If a new thread is switched in during a transaction, the transactional state in the cache is no longer meaningful since it belongs to the previous thread.

The context switch problem is not an issue for normal memory operations since the cache is completely abstracted away from the thread in a non-transactional context. However, In the case of the transactions a context switch separates the transaction from its speculative state and thus requires an abort.

3.3 Overcoming Size and Length Limitations in Hardware

It is conceivable that HTM can be extended to support large and long transactions but such a system does not currently exist. In fact, [6] suggests one such design. However, even that design

has not been fully worked out and has its problems. I believe that there is another more general approach to hardware transactional memory that does not depend on the cache. However, it is unclear what that system will look like as such a design is not trivial.

Therefore, it is safe to consider the problem of size and length limitations as being largely unsolved for hardware transactions. For this reason, these limitations are fundamental to any hardware scheme, not just HTM.

4 STM: Software Transactional Memory

Transactional memory can be implemented completely in software. The FLEX software transaction system [1] (I will call it STM for the remainder of this paper), is one such system. I will use STM as the baseline software transactional memory system since it is available and is currently one of the more efficient software transaction designs [1].

Figure 5 shows the object structure used in STM. Each object has, in addition to its normal fields, a `versions` pointer and `readers_list` pointer. `versions` points to commited versions of the object and possibly a transactional version as well. `reader_list` points to a linked list of all the pending transactional readers of the object. Each version is linked to a transaction tag that is specific to each transaction.

STM stores all transactional data in a clone (or version) of the object. When a transaction writes a field, a clone of the object is made (if not done so before). Then the value of the field in the original object is set to some predetermined constant `FLAG`. Then the new transactional value is written to the live transactional version of the object. The `FLAG` value is written to indicate to other transactions that the field is being held transactional. In addition, the writing transaction adds itself to the `readers_list` of the object as well.

When a transactional read is performed, STM simply adds the reading transaction to the `readers_list` of the object and reads the value of the field. The `readers_list` indicates to other transactions that a field in the object is being read by a pending transaction.

When a transaction is committed or aborted, the transaction tag is simply changed. Therefore, all objects linked to the transaction tag will be seen as either commited or aborted data (respectively) in one short operation.

To detect conflicts, STM adds a check on each transactional read and write operation:

- On each read STM checks if the field being read is marked `FLAG`. If so, another pending transaction may be writing to it. In this case, the reading transaction walks the `versions` list and aborts the pending transaction. However, it may also be the case that another transaction wrote to it previously and has since committed. In this case, the reading trandsaction walks the `versions` list, reads the correct value from the committed version and writes it back into the original version.

- On each write STM checks if the object has any readers in the `reader_list`. If so, it walks the `readers_list` and aborts all pending readers. The transaction then performs the transactional write as described previously.

STM is efficient since it can take advantage of many compiler optimizations. Many of the checks can be avoided if the compiler determines they are unnecessary. For example, if a transaction reads

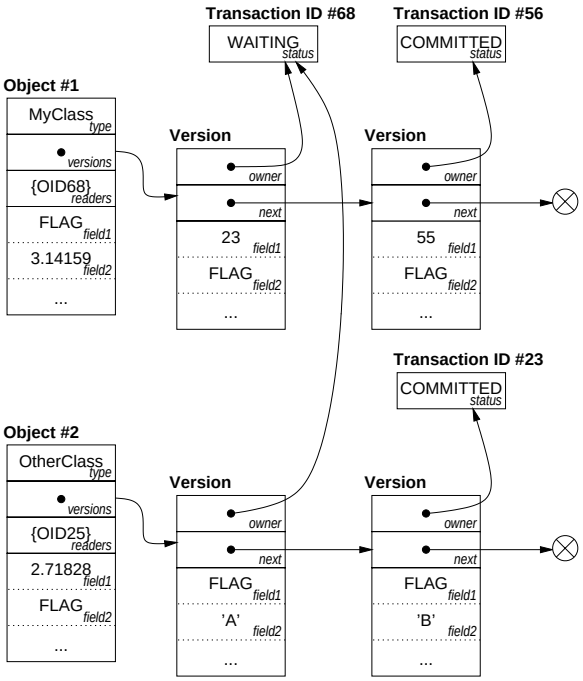

Figure 5: Object structurte in STM. Each object has, in addition to its normal fields, a `versions` pointer and `readers_list` pointer. `versions` points to commited versions of the object and possibly a transactional version as well. Each version is linked to a transaction tag.

from a field it previously wrote to, it can just read the value from the transactional version of the object directly. It does not need to read the field from the original object and check that it is marked `FLAG`. Such optimizations actually decreases STM overhead for long transactions [1] since long transactions tend to access the same fields often.

5 The Case for Hardware-Software Integration

To justify hardware-software integration, I will first argue that a "bounded" approach such as HTM is undesirable. Then I will argue that integrating hardware and software is a sensible and very natural approach to overcome the limits of HTM.

5.1 The Case for Boundless Transactions

In this section, I argue that boundless transactions are desirable. I first consider the size limitation and then the length limitation. Much of this section was taken from the ISCA submission on Boundless Transactional Memory [6].

First, consider the size limitation problem from the point of view of the compiler. Suppose that the transactional memory system provides support for transactions that touch, say, 1000 cache lines. What should the compiler do if it needs to produce a transaction that is larger than 1000 cache lines? Quit and tell the programmer that her transaction is too large? And, how does the

compiler know how big the transaction really is, especially if it contains loops or subroutine calls? Finally, how does the compiler cope when the bound on transaction size might vary from machine to machine? Does the transaction bound become a fixed architectural parameter that dictates machine type so that a binary with one transaction bound won't run on a machine with a different parameter?

Further, as mentioned in Section 3.1, the size limitation is not simply a fixed number. The size limit depends on many different cache parameters that make it very difficult to predict. Therefore, the compiler will not even have a simple number such as 1000 that it can use to limit transaction size.

Therefore, it makes sense to view transaction size as an implementation parameter, like cache size or memory size, which can vary without affecting the portability of binaries. Code should not need to be recompiled to deal with a different bound on transaction size. From this point of view, boundless size transactions are a necessity.

Secondly, consider the length limitation. As with size, considerations such as modularity and compiler complexity dictate that a set length limitation is undesirable. For example, context switches may prevent long transactions from ever completing. It may be true that most transactions will be short and thus will likely not be interrupted by a context switch. However, in modern systems, context switches can occur quite often at regular intervals. Therefore, a long transaction that does not fit within its operating system time-slice, for example, may never commit in a bounded system.

In addition, there are cases of context switches that occur much more frequently and are far less predictable. TLB exceptions is an example. TLB refills may occur quite often especially when operating on large data sets. Therefore, like the size limitation, the hardware length limitation is truly a serious problem that needs to be addressed.

5.2 The Case for Integrating With Software

To understand whether software is a viable solution to the limitations in HTM, we must understand the the trade-offs between HTM and STM more quantitatively. I measured the overheads incurred by HTM and STM for two different conditions using a benchmark based on the `node_push` example in Section 2; the "worst" case and the "more realistic" case. The "worst" case is when we have very small transactions running back-to-back. This case should give an upperbound on transaction overhead since any per-object overheads cannot be amortized over the length of the transaction. In addition, back-to-back short transactions puts the most pressure on the processor pipeline in HTM. The processor needs to have several active transactions to sustain performance. Though small transactions are expected to the common case, they will likely not be run back-to-back since most transactions require some pre-processing. The "more realistic" case describes this situation where there is some non-transactional processing between small transactions. The processing in for `node_push` is simply picking the two nodes to be used next.

HTM has very low overhead. As shown in Figure 6, HTM is has only a 1.5x slowdown over the base and is 3.3x faster than using locks even in the "worst" case. In the "more realistic" case, HTM performs almost as well as the base (only 4% overhead).

On the other hand, STM has much higher overhead. As shown in Figure 6, in the "worst" case STM is almost 12x slower than HTM. In the "more realistic" case, some of that overhead can be hidden. However, even then STM is almost 2x slower than HTM.

Atomicity Mechansim	"Worst"	"More Realistic"
	Cycles (% of Base)	
Locks	505%	136%
HTM	153%	104%
STM	1879%	206%

Figure 6: The overhead associated with locking, HTM, and STM for the `node_push` benchmark. The Base case is the `node_push` subroutine without any synchronization. The "worst" case is when each small transaction (`node_push`) is run back-to-back. The "More Realistic" case is when there is some computation between small transactions (`node_push`). The computation between transactions was the calculation of the random nodes to pick next.

Given the extremely high overhead of STM, it is clear that we do not want to use only STM for all transactions. Doing so would overcome any size and length limitations, but its performance would pervent it from being practical. In addition, we need to consider what transactions will typically look like. In the common case, transactions will be small and short. Therefore, STM would be incurring a high overhead for all transactions when all but a few would have worked in HTM with much less overhead.

Ideally, we would like to run all transactions in hardware since HTM has very low overhead however this is not possible because of hardware limitations. Therefore, it is desirable to run as many transactions as possible in hardware. In the case where the transaction does not fit in hardware, we wish to switch to something else that can handle the large transaction correctly.

STM can handle large and long transactions correctly. Therefore, switching from HTM to STM is a possible solution. However, is STM the best choice given it has such high overhead?

Firstly, in an integrated hardware-software approach, if the common case performance is good, we can afford to have a higher overhead in the uncommon case without drastically affecting overall performance. Thus, even though STM overhead is high, we expect large and long transactions to be uncommon. Therefore, even if they incur a high overhead, the overhead will be amortized over the common case.

Secondly, as discussed in Section 3.3, there are no known hardware transactional memory schemes that are boundless. Since the only way to achieve boundless transactions is in software at the moment, an integrated approach is a very natural solution to problem. In the common case, we get small transactions that run fast in HTM. In the uncommon case, we get large transactions that are slower but still work.

Given that software is a good alternative, STM is a good choice. As discussed in Section 4, STM is one of the most efficient software transaction systems available. STM has lower overhead for large and long transactions which is exactly where STM is intended to be used.

6 HSTM: The Integrated Hardware-Software Approach

The integrated hardware-software scheme (HSTM) runs all transactions in hardware first and switches to software if the hardware transaction aborts. In this section, I will describe the changes to HTM that are necessary to make such an integrated approach work.

6.1 Software-Compatible HTM

For HSTM to function correctly, both hardware and software transactions must interact correctly with one another. Since transactions in HSTM switch from hardware to software, a system may have both hardware and software transactions operating at the same time. Therefore, it is essential that HSTM ensures that all transactions achieve atomicity with respect to all other transactions (both hardware and software).

Therefore, HSTM must be able to detect conflicts between all transactions. If a hardware transaction conflicts with another hardware transaction, the normal HTM mechanism will detect the conflict. Similarly, software-software conflicts can be detected using the normal STM mechanism. Therefore, the only added functionality is the ability to detect and handle conflicts between hardware transactions and software transactions. I will describe how HTM is changed so that these conflicts can be detected and how they are dealt with.

6.1.1 Conflict Detection

If a software transaction touches a location owned by a pending hardware transaction, the conflict is detected and handled by the standard HTM and STM mechanism. Software transactional read and write operations involve respective load and store memory operations to the object field. Therefore, to the hardware transaction, this appears the same as a normal conflicting memory operation. These conflicts are detected using the cache-coherency mechamism as in HTM.

If a hardware transaction touches a location owned by a pending software transaction, the conflict may not be detected by the standard HTM and STM mechanism. I add some additional checks to the hardware transaction to detect and handle these conflicts correctly. These checks are shown in Figure 7 and outlined below:

- **Load Check:** On each transactional load instruction, the hardware transaction must check whether the value loaded is `FLAG`. If so, there is a potential conflict since the `FLAG` value may indicate that a pending software transaction has written to that field. Therefore, if the value loaded is `FLAG`, the hardware transaction is aborted. The load check is performed in software.

- **Store Check:** On each transactional store instruction, the hardware transaction must check whether the object is being read by any pending software transactions. This is done by checking if the `readers_list` pointer is `NULL`. If `readers_list` is not `NULL`, there is a potential conflict and the hardware transaction is aborted. Like the load check, the store check is performed completely in software.

To show that these two checks are sufficient to detect all such conflicts, I consider all 4 possible cases:

- **Software transaction first reads field x:** The software transaction adds itself to the `readers_list` of the object. Field x is not marked with `FLAG`. We assume that this is the only software transaction operating on the object.

  - **1. Hardware transaction then reads field x:** The hardware transaction loads the value from field x and checks if it is `FLAG`. Since it is not `FLAG`, the hardware transaction continues normally. This is correct since two transactions can be reading the same object field at the same time.

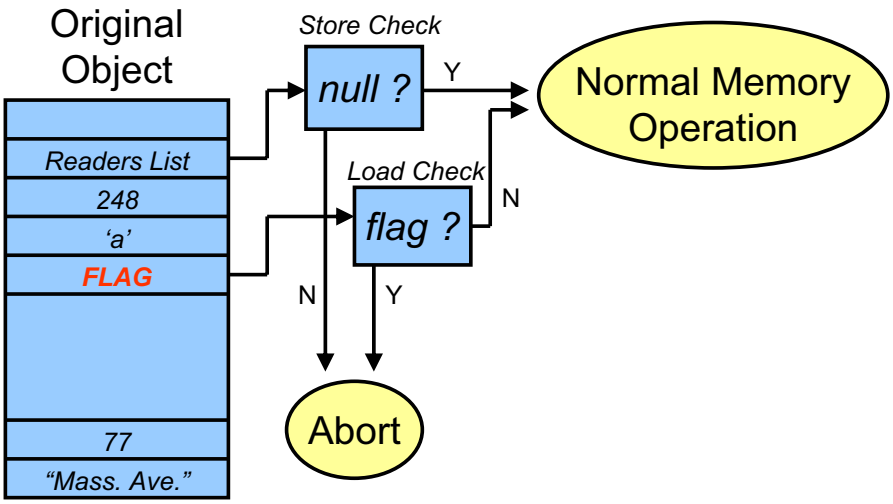

Figure 7: Additional checks performed on HTM load and store operations that are required to be software-compatible. These checks are performed in software, not hardware. For the load check, if the field being read is flagged, the hardware transaction is aborted. Otherwise, the hardware transaction proceeds normally. For the store check, if the readers list of the object not empty, the hardware transaction is aborted. Otherwise, the hardware transaction proceeds normally.

  - **2. Hardware transaction then writes field x:** The hardware transaction checks if `readers_list` is `NULL`. Since the software transaction already added itself to the `reader_test`, this check will fail indicating a conflict. Then the hardware transaction will be aborted.

- **Software transaction first writes field x:** The software transaction writes `FLAG` to field x and makes a transactional version of the object. The software transaction also adds itself to the `readers_list`. The new value is then written to the transactional version.

  - **3. Hardware transaction then reads field x:** The hardware transaction loads the value from field x and checks if it is `FLAG`. This check returns positive indicating there is a conflict. Then the hardware transaction is aborted.
  - **4. Hardware transaction then writes field x:** A conflict will be detected in the same way as in case 2. Then the hardware transaction is aborted.

6.1.2 The `xABORT` Instruction

Since all the additional HSTM checks are performed in software, a hardware transaction in HSTM requires the ability to abort itself if these checks indicate a conflict. This functionality is provided by adding the additional instruction `xABORT` to HTM ISA. `xABORT` simply aborts the current running transaction.

Adding `xABORT` to an HTM-capable processor is a relatively simple operation. The abort is performed at the same point in the processor pipeline as transactional commits. In processors that perform transactional commits at instruction graduation, the `xABORT` instruction needs to be graduated in instruction order. On graduation, the processor simply aborts the current running transaction.

The `xABORT` instruction changes the properties of hardware transactions in a very subtle way. Namely, hardware transactions are no longer non-blocking. Once the transaction is given the ability to commit suicide, one large transaction may block the completion of other transactions. This a not desirable but since HSTM eventually falls back to a non-blocking software scheme, the entire system is still non-blocking.

6.1.3 Time and Space Overhead

Software-compatible HTM does not come without a cost. Firstly, the added checks for every memory operation add some overhead on runtime. In the "worst" case, these checks result in a 2.2x slowdown over standard HTM (1.1x in the "more realistic" case). This overhead is expected since each transactional store operation now requires an additional load to read the `readers_list`. In addition, there is an additional conditional jump for every memory operation. Fortunately, the jump is very predictable since conflicts are uncommon.

Secondly, there is some space overhead required since each object requires the added `readers_list` and `versions` pointers. These additional fields are required even when running in pure HTM-mode since another processor may be operating on this node in software. Since most transactions will be run in HTM-mode, the additional space overhead is not needed most of the time. However, as shown in [1], the added overhead is quite reasonable for all applications in the SPECjvm98 benchmark suite.

7 Evaluation

For evaluation, I implemented the a benchmark application based on the `node_push` example given in Section 2. In this section, I evaluate how well HSTM performs on this benchmark. First I explain the implementation of NSTM and the simulation environment used. Then I give the performance results from running the `node_push` benchmarks on HSTM.

7.1 Simulation Environment

I used the UVSIM software simulator for all the implementation and evaluation. UVSIM was developed at the University of Utah and is based on the URSIM [8] simulator. UVSIM models a shared-memory multi-processor system similar to the SGI Origin 3000. UVSIM contains an extremely accurate model of the MIPS R10K [7] core at each node in the network. Cache coherency is maintained using a directory-based scheme.

I implemented a complete HTM system in UVSIM. The HTM implementation involved changes to the cache, cache controller, and processor pipeline models. I also added the `xABORT` instruction as discribed in Section 6.1.2.

I implemented a subset of STM functionality in the `node_push` benchmark application. Unfortunately, full STM functionality was not available in the FLEX compiler at this time. Therefore,

to get accurate performance numbers for STM and HSTM, I hand-coded the software transactions structure into the benchmark appliaction. Since I was only interested in evaluating overhead, I only implemented the subset of STM that would be used in my tests.

I implemented the changes to HTM in the same way; by hand-coding the checks into the benchmark application.

7.2 Size Performance

To evaluate how HSTM performs for various size transactions, I changed the `node_push` benchmark. The benchmark was modified to touch a variable number of nodes, as opposed to just two. Nodes were touched in sequential memory order one after another.

Figure 8 shows the performance of pure hardware and software transactions. Hardware transactions stop fitting after 1700 nodes. However, even though software is about 2.3x slower, software trandsactions continue to handle all the larger sizes.

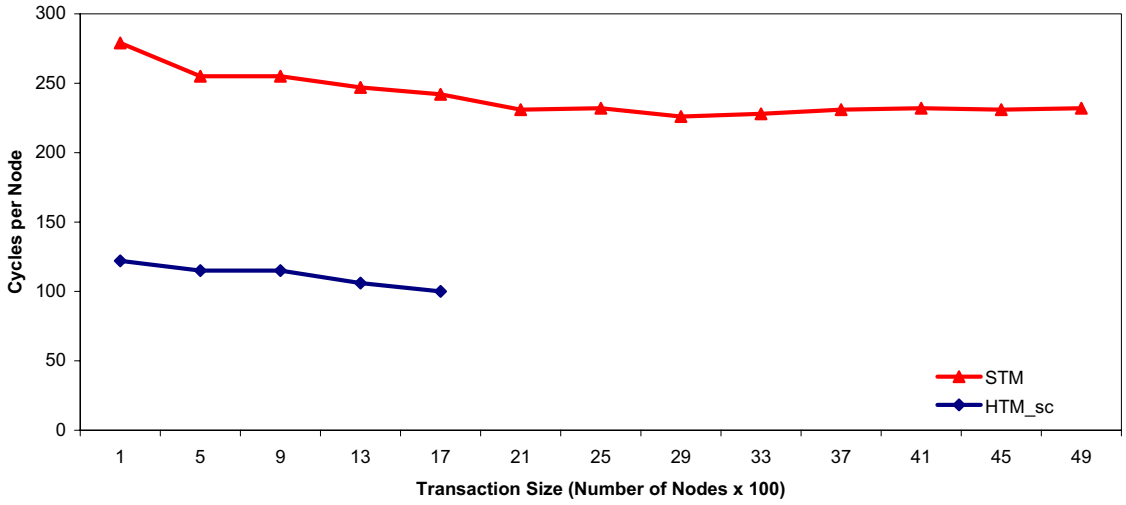

Figure 8: The performance of pure hardware and software transactions for different size transactions. HTM_sc is the modified software-compatible hardware transaction. The hardware transactions fail for all transaction sizes greater than 1700 nodes.

Figure 9 shows the performance of HSTM compared to pure hardware and software transactions. HSTM performs almost exactly the same as HTM for transactions that fit in the hardware (the HTM line is completely covered by the HSTM line). Such performance for small transactions is exactly what we want since it represents common case performance. For transaction sizes greater than 1700 nodes, the transaction can no longer fit in the hardware. HSTM then falls back to software to handle these larger transactions. The overhead associated with the first try in hardware is very visible at the transition point. However, even then HSTM is only $\tilde{3}5\%$ slower than STM. In addition, as the transaction size increases, the overhead decreases as it is amortized over more

elements.

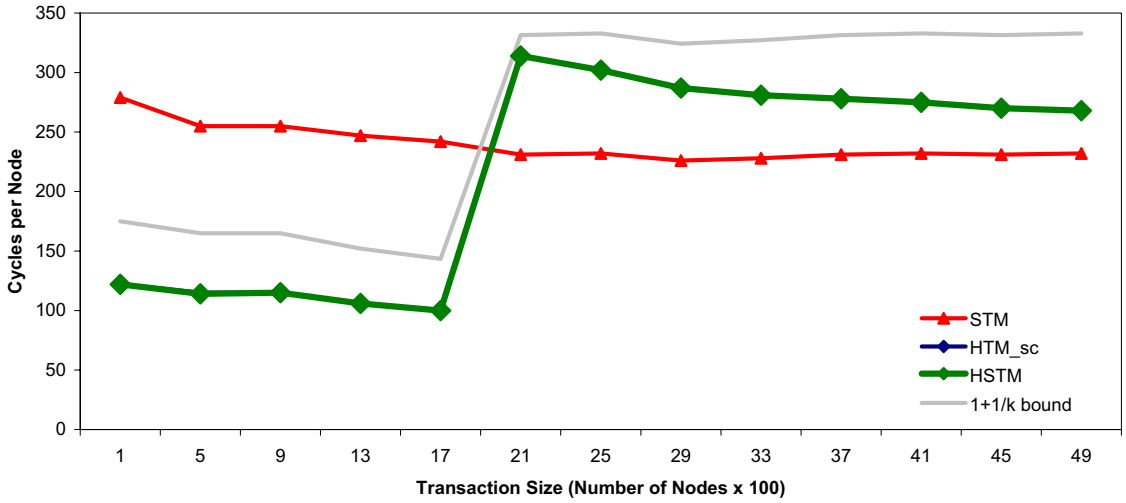

Figure 9: The performance of HSTM compared to pure hardware and software transactions. HTM_sc is the modified software-compatible hardware transaction. The analytical upper bound $(1+1/k)$ is also shown for $k = 2.3$.

I did some analysis to find the worst case performance of HSTM for different size transactions. The HSTM switching strategy has an analytical competitive ratio of $1+1/k$ over the optimal choice (where k is the slowdown of software over hardware). This is always true regardless of the length of the transaction and the capacity of the cache. See Appendix B for more details on the analysis. Though this switching strategy is a very simple, it gives a very good competitive ratio. Notice that it is always within 2x of optimal. From the simulation results, k was found to be $\tilde{2}.3$ and the upper bound is shown in Figure 9.

7.3 Length Performance

To evaluate the performance of HSTM for various length transactions, I modified the `node_push` benchmark. Instead of changing the length of the transactions, I changed the probability that a transaction would be aborted. This was done since UVSIM does not have an operating system that performs context switches at regular intervals.

To vary the probability of abort, I changed the probability that the transaction would case a TLB exception. I modified the `node_push` benchmark to pick from a variable size pool of nodes. Since nodes are picked at random, the node pool size is inversely related to the probability the picked node will miss in the TLB. A TLB miss causes a context switch to the software handler that walks the page table.

Figure 10 shows the performance of pure hardware and software transactions for different abort probabilities. The bars essentially show the expected value of the number of tries in hardware

before a successful commit. Since we are picking two nodes, the expected value increases as the square of the number of nodes in the pool. For that reason, runtime of the hardware transactions increases drastically when aborts become more frequent. Hardware is much faster than software when context switches are unlikely. However, at 22000 nodes, software actually starts to become faster than hardware since hardware transactions are tried approximately 2 times before they commit successfully.

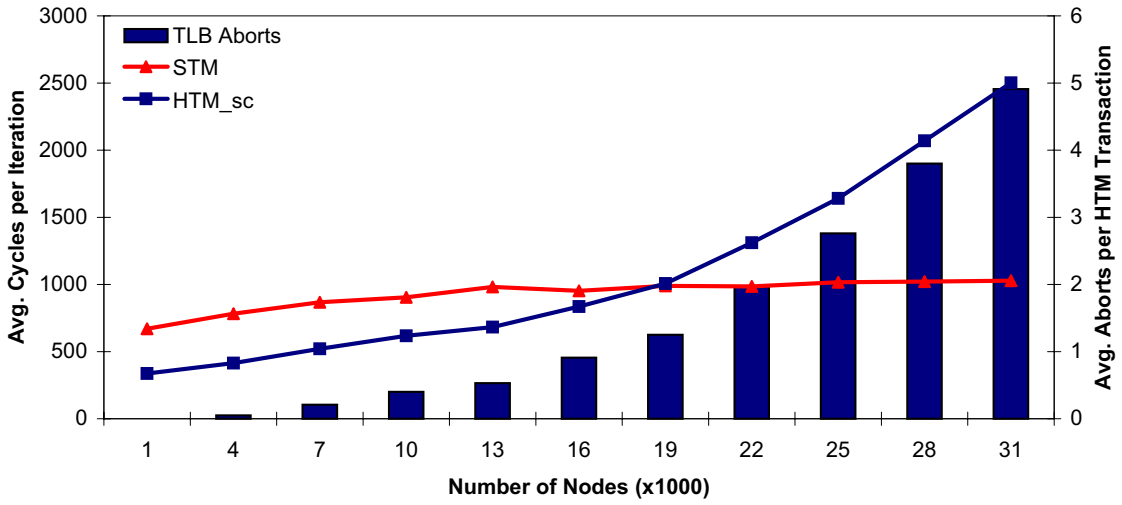

Figure 10: The left y-axis and the lines show the performance of pure hardware and software transactions. The right y-axis and the bars shows the average number of aborts per successful transaction. HTM_sc is the modified software-compatible hardware transaction.

Figure 11 shows the performance of HSTM compared to pure hardware and software transactions. As with size, the HSTM line tracks the lower of the two other lines pretty well. In fact, the worst case overhead is only $\approx 40\%$. In most cases, the overhead is only $\approx 25\%$.

8 Conclusions

Hardware transactional memory systems such as HTM have very serious limitations that can be easily overcome using software. An integrated design such as HSTM can take advantage of this fact and allow common case small transactions to run fast while still providing support for large transactions in the uncommon case.

This paper has described one such integrated approach; HSTM. HSTM is simple to implement but still provides good performance. In the absence of better hardware transactional memory systems, an integrated approach such as HSTM is a good alternative.

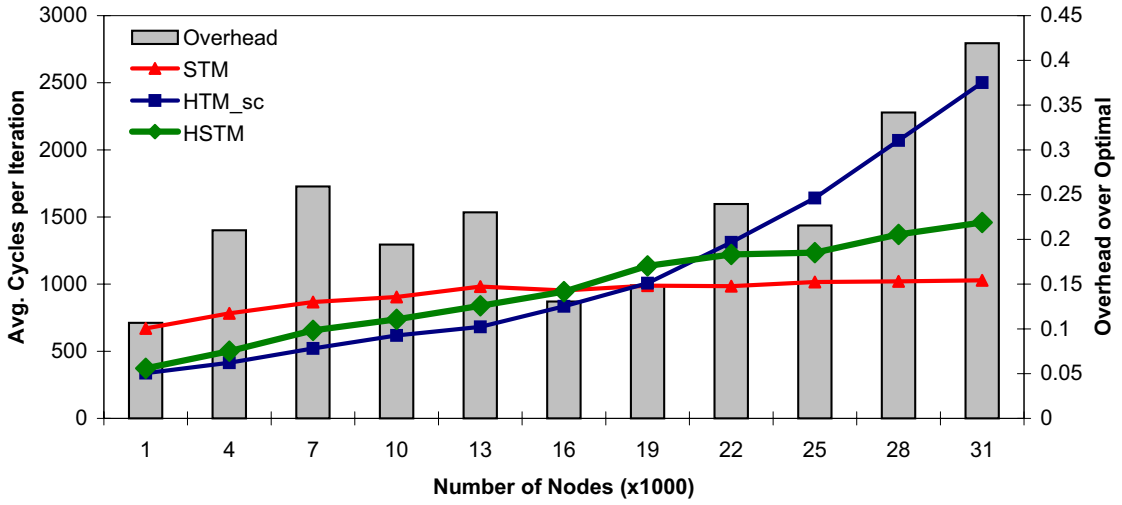

Figure 11: The left y-axis and the lines show the performance of HSTM compared to pure hardware and software transactions. The right y-axis and bars shows the HSTM overhead over the optimal (the lower of HTM_sc and STM). HTM_sc is the modified software-compatible hardware transaction.

References

[1] Ananian, C. S. and Rinard, M. [2003]. *Efficient Software Transactions for Object-Oriented Languages*, MIT CSAIL, Unpublished.

[2] Herlihy, M. and Moss, J. E. B. [1992]. *Transactional Memory: Architectural Support for Lock-Free Data Structures*, Technical Report 92/07, Digital Cambridge Research Lab, Cambridge, Massachusetts.

[3] Herlihy, M. and Moss, J. E. B. [1993]. *Transactional Memory: Architectural Support for Lock-Free Data Structures*, Proceedings of the 20th Annual International Symposium on Computer Architecture.

[4] Knight, T. [1986]. *An architecture for mostly functional languages*, Proceedings of the 1986 ACM Conference on LISP and Functional Programming (LFP), pages 105-112. ACM Press.

[5] Knight, T. [1989]. *System and method for parallel processing with mostly functional languages*, United States Patent 4,825,360.

[6] Leiserson, C. E. et al. [2003]. *Hardware Support for Boundless Transactions*, MIT CSAIL, Unpublished.

[7] Yeager, K. C. [1996]. *The MIPS R10000 Superscalar Microprocessor*, IEEE Micro Volume 16 Issue 2, 28-41.

[8] Zhang, L. [2001]. *URSIM Reference Manual*, Technical Report UUCS-00-015, University of Utah, Salt Lake City, Utah.

Appendix

A Additional Material

All additional material used for this paper such as source code can be found at http://slie.ca/projects/6.895/.

B Space Limitation Analysis

In this section, I look at the general problem of space limitation. The goal is to find a bound on the how well HSTM's switching performs against optimal. In addition, I provide an alternative switching strategy and analyse its performance.

I model the problem as follows:

- Cost is the number of memory references x.
- The size of the hardware transactional buffer is b. If $x \leq b$, then the transaction can be run completely in hardware with cost x.
- The cost to run a transaction completely in software is kx where $k > 1$.
- A transaction must be run completely in hardware or completely in software. Though we think there are ways that we can work around such a constraint, this represents a simple transition from hardware to software. In other words, we assume that a transaction cannot run partly in hardware and partly in software. Instead, if we wish to switch, we must abort the hardware transaction and restart it in software.
- During execution of the transaction, we know b and we know the number of transactional memory references we have already made. However, we do not know x. This is reasonable since the hardware can provide information such as the size of it's transaction buffer and how full it is. However, when we are running there is generally no way for the program to know how much further it needs to run until it finishes. We will relax this constraint later.

Given this model, I wish to find the optimal time to switch to a software transactional memory system.

B.1 The Straight-Forward Approach

The straight-forward approach (used in HSTM) is to simply run the transaction completely in hardware if possible. If we run out of space, we just switch to software. To evaluate how good this scheme is, we compare its worst case to the best possible off-line scheme as in the ski rental problem. There are two possible cases: 1) $x \leq b$ and 2) $x > b$.

Case 1 ($\boldsymbol{x \leq b}$): In this case, the optimal off-line scheme has cost $C_{opt} = x$ and the online scheme has cost $C = x$ as well. So we get a competitive ratio of $R_c = C/C_{opt} = 1$.

Case 2 ($x > b$): In this case, the optinal off-line scheme would just start off in a software transaction so it would have cost $C_{opt} = kx$. The online scheme would run completely in hardware and then have to restart in software. The worst case occurs when x is just slightly more than b. In this case, we would get a cost of $C = x + kx$. Therefore, the competitive ratio is $R_c = C/C_{opt} = 1 + 1/k$.

Since we are concerned with the worst-case, the straight-forward approach has a competitive ratio of $1 + 1/k$. This is pretty good since it has an upper bound of 2 however can we do better?

B.2 Doing Better Than $(1 + 1/k)$-Competitive

To show that we can actually do better than $1 + 1/k$, we use a scheme similar to the ski-rental problem. We start off running in hardware. Then at a certain point in time, we make a decision as to whether or not to switch to software. We consider only one decision point right now.

We set the switch point to be ab where $0 \leq a \leq 1$. If we have made ab transactional memory references and we are not done, we switch to software with probability p. We continue running in hardware with probability $1 - p$. As before, we have two cases.

Case 1 ($x \leq b$): In this case, the worst case will be when $x > ab$ since otherwise we will run with a competitive ratio of 1 since $C_{opt} = x$. So if $x > ab$, we will need to make a decision at ab. With probability $1-p$ we continue in hardware and the final cost is just x. With probability p we switch to software and the final cost is $ab+kx$. This yields a total expected cost of $C = (1-p)x+p(ab+kx)$. From this equation we see that the worst case occurs when x is slightly more than ab since the ratio of hardware to software cost is highest then. This gives a total cost of $C = (1-p)ab + p(ab + kab)$ and a competitive ratio of $R_c = C/C_{opt} = 1 - p + p(k + 1)$.

Case 2 ($x > b$): In this case, the optimal off-line cost is simply $C_{opt} = kx$ since we just start start running the transaction in software from the beginning. In the online scheme, with probability $1-p$ we run until we fill hardware and then switch to software. This case has a final cost of $b+kx$. With probability p, we run until ab and then we switch to software. This case has a final cost of $ab + kx$. Therefore, the final expected cost is $(1-p)(b+kx) + p(ab+kx)$. From this equation, we see that the worst case occurs when x is just slightly more than b since the ratio of hardware to software cost is highest then. Given that, we get an expected cost of $C = (1-p)(b+kb)+p(ab+kb)$ and a competitive ratio of $R_c = C/C_{opt} = \frac{1}{k}((1-p)(k+1) + p(a+k))$.

Since we are concerned with worst-case, we want the competitive ratios from both parts to be equal. Otherwise, the competitive ratio would just be the higher of the two. If we set them equal and solve for p we get $p = (k^2 + 1 - a)^{-1}$. By substituting this back into the competitive ratio of either case, we get an overall competitive ratio of $1 + k(k^2 + 1 - a)^{-1}$. We note that the choice of a affects this competitive ratio. However, since we want a small competitive ratio, we want to choose the smallest possible a ($a = 0$). We must remember however that our competitive ratio from case 1 was obtained by dividing by ab and so $a \neq 0$. However setting $a = 0$ is ok here since we were actually a bit sloppy in case 1. Namely, we wanted x to be slightly higher than ab, not ab exactly. Therefore, with $a = 0$, we have a competitive ratio of $1 + k(k^2 + 1)^{-1}$.

It should be pointed out that $a = 0$ means that our decision point is right at the beginning before we even start executing the transaction. Though this may seem slightly odd, it actually makes sense since we are concerned with worst-case only. In other words, since the worst-case is

always going to be after the ab point (before that point always gives us a competitive ratio of 1), we might as well make the decision right at the beginning and not waste the time doing work in hardware and then throwing it away to start all over in software.

This result implies that knowing the size of the hardware buffer b does not help us since we want to make the decision right at the beginning. This is very desirable since the algorithm can be oblivious to the hardware constraints while still achieving good performance.

Finally, we note that the competitive ratio $1 + k(k^2 + 1)^{-1}$ is less than that obtained in the straight-forward case of $1 + 1/k$. For example, with $k = 1.2$ the new scheme gives a competitive ratio of ≈ 1.49 where the probability $p \approx 0.41$. On the other hand, the straight-forward scheme gives a competitive ratio of ≈ 1.83.

B.2.1 Is This Really Better?

The randomized switch strategy described here was not used in HSTM. There were two main reasons for this decision.

Firstly, as shown in Figure 9, we are rarely operating at the worst case. In other words, the real world is not a game where the inputs are set up by an adversary.

Secondly, for the value of k in HSTM ($k \approx 2$), the difference between the two schemes is only approximately 15% in the worst case. In addition, the analysis did not take into account the fact that making a random number function call is not free. For the `node_push` benchmark, the transactions are small enough such that making the random choice adds a significant amount of overhead not modeled in the analysis.