A Software Transactional Memory System for an Asymmetric Processor Architecture

Felipe Goldstein, Alexandro Baldassin, Paulo Centoducatte, Rodolfo Azevedo
Institute of Computing
University of Campinas – UNICAMP
felipe.goldstein@students.ic.unicamp.br, {baldas,ducatte,rodolfo}@ic.unicamp.br

Leonardo A. G. Garcia
Linux Technology Center
IBM
lagarcia@br.ibm.com

Abstract

Due to the advent of multi-core processors and the consequent need for better concurrent programming abstractions, new synchronization paradigms have emerged. A promising one, known as software transactional memory (STM), aims to use transactions as the key synchronization mechanism to ease program development as well as increase its performance. Many (if not all) of the current STM implementations target homogeneous architectures. In this paper we describe an implementation of an STM system for an asymmetric architecture, the Cell BE. We evaluated our Transactional Software Cache (TSC) mechanism using a well-known micro-benchmark (IntSet) and the Genome application from STAMP. The results show that an STM implementation for the Cell architecture is feasible if the shared-memory programming model is adopted. When compared to a conventional lock-based implementation, the STM version of Genome obtained a performance gain of 84% and 24% with large and small input sets, respectively.

1 Introduction

For many years the increase in clock frequency has made it possible for each new generation of microprocessors to become faster. Unfortunately, this trend has come to an end since increasing the clock speed causes a tremendous power dissipation which cannot be remedied by current cooling technology. In order to make use of the ever-increasing number of transistors the semiconductor industry has chosen to integrate multiple cores of execution into a single chip, given rise to the multi-core era. Although most of these multi-core processors have homogeneous architecture, there are some work towards asymmetric models, like the Cell Broadband Engine (Cell BE) [9].

The shift towards parallel computing has an enormous impact on software development. The general consensus on shared memory architectures is that writing efficient and scalable code using traditional lock-based approaches are very difficult due to their low abstraction level and lack of composability [14]. A promising alternative is known as Transactional Memory (TM) [10], wherein transactions are specified by programmers and the system is responsible to execute them atomically and in isolation. Proposals for TM implementation have emerged in three fronts: Hardware (HTM), Software (STM), and Hybrid (HyTM).

In this paper we describe an implementation of an STM system for an asymmetric architecture, the Cell BE. To the best of our knowledge this is the first implementation of an STM in a heterogeneous multicore processor. Although we are targeting one specific architecture, we believe that the techniques employed here would be useful for similar machines. One of the main differences between a Cell BE processor and other general purpose processors is that the programmer has deep access to the processor internals. Therefore, it is a programmer responsibility not only the control of all the communication mechanisms provided by the hardware (i.e. DMA, mailboxes, and signals) but also the management of the non-cached local store memory available in each of its synergistic cores (SPE).

We evaluated our Transactional Software Cache approach, called TSC, through the IntSet [8] microbenchmark using two different data structures (linked-list and hashtable) and the Genome application from STAMP [3], allowing us to assess our implementation with a robust application. We compared the results against a global locking scheme and concluded that our approach is more effective. The Genome application using our TSC approach obtained a performance gain of 84% with a large input set and a gain of 24% with a small input set when compared to the lock-based version, also achieving a better scalability. The main contributions of this paper are: (1) The first implementation of an STM system on top of an asymmetric processor architecture. (2) A software cache with consistency for the Cell BE expanding the one provided with the current IBM SDK for Multicore Acceleration 3.0 [1].

This paper is organized as follows: Section 2 presents a brief overview of the Cell BE processor. Section 3 describes our software transactional memory infrastructure and how it deals with the Cell BE asymmetry. Section 4 shows our experiments and presents our preliminary results. Finally Section 5 describes related work and we conclude in Section 6.

2 The Cell BE Processor

The Cell Broadband Engine Architecture introduced a new organizational approach to processor design when it was first released in 2006. The processor is composed by a Power Processing Element (PPE) and eight Synergistic Processing elements (SPE). Each SPE has its own local store (256K) which is separated from system memory. While the PPE can access system memory via load and store instructions, the SPE cores access it through a globally-coherent DMA engine. Furthermore, the instruction set of the PPE is different from that of the SPE. This new organization made it possible to have supercomputer processing power in just one chip.

Although the initial purpose of the Cell BE processor, the first real incarnation of the Cell BE Architecture, was to have outstanding computing performance, specially on game and multimedia applications, and real-time responsiveness to the user and the network [9], the Cell BE development community was rapidly enlarged by people seeking new opportunities for the huge computation power and unique design provided by the new architecture.

One important difference between conventional processors and the Cell BE is the fact that only a single program context is supported at any time on an SPE core. Although this can sound a bit strange at first as this kind of limitation is not seen in conventional processor for a long time, this unique design decision simplified lots of aspects related to the SPE programmability, making the SPE the perfect choice for computational demanding tasks that should not be interrupted or for other types of tasks that need some level of isolation from the outside world.

Programming the Cell BE is complicated by several factors. On the one hand, programmers must deal with the communication aspects among the cores and system memory (such as issuing DMA commands). On the other hand, they also need to manage the different ISAs and distribute the work so as to make the most out of the architecture. With time, all the low level details are expected to be added on a compiler or a specific library that deals with the ins and outs of a specific problem scenario, making the development task easier. However, the current stage is quite far from that reality in a number of research fields.

3 Transactional Memory and CellBE

The transaction concept is the key abstraction element in STM systems and the one programmers reason about. In its simplest form, a transaction is just a block of instructions guaranteed to execute atomically and in isolation from the rest of the system. Transactions aim to provide synchronization abstractions as easy to program as with coarse-grained locks and, at the same time, perform as well as fine-grained lock-based synchronization.

Transactional memory systems transfer the burden of concurrency control from the programmer to the underlying system implementation. As multiple transactions are speculatively executed, the system keeps track of every memory location accessed by each transaction. Both old and new data are implicitly managed (data versioning) and conflicts among transactions can be safely detected. In case a conflict arises it is possible, for instance, to roll back a transaction (using the old data) without any intervention from the programmer side.

Our STM model is based on the TL2 [4] algorithm adapted to suit the Cell BE architecture needs as will be explained shortly. The idea behind the TL2 algorithm can be briefly described as follows: A global version-clock is maintained in the system for consistency. Each memory location is augmented with a lock and a version number (called henceforth the versioning directory). When a transaction starts executing, it reads and saves a local copy of the version-clock. Every memory location read by a transaction must be first validated against the transaction version-clock by using the corresponding version number. The set of all values read is called the transaction read-set. Similarly, the write-set is comprised of all memory locations written and buffered internally by a transaction. When a transaction is about to commit it must acquire locks for each location in its write-set, increment the global version-clock in main memory, validate the entire read-set, write back the write-set to main memory with appropriate version numbers, and finally release the write-set locks. For further details please refer to the original TL2 algorithm in [4].

When adapting the TL2 algorithm to the Cell processor we focused on the simplicity and flexibility of the model to take advantage of the architecture particularities in a way to facilitate its implementation. This preliminary version is a proof of concept of how the software cache model and software transactional memory together can solve many problems faced by programmers when developing a shared-memory multithreading application for the Cell BE processor. In the rest of this section we show how these two concepts can work together to make programmers life easier.

3.1 Conventional Software Cache

Our STM architecture leverages the conventional software cache implementation that comes with the IBM SDK. This software cache is implemented as an array of cache lines and a directory array. Both structures are stored in the SPE's local store. Each cache line contains a *Tag*, a *Valid* and *Modified* bits, and 128 bytes of data. In our experiments we use a 4-way associative cache configuration. The implemented replacement policy is Round-Robin.

The software cache API is exposed to the user via a set of macros. These macros take care of eventual DMA transfers when system memory is referenced, thereby hiding the communication complexity from programmers. The macros are of the form:

- `LOAD(effective_address)`: returns the word in main memory pointed by `effective_address`;
- `STORE(effective_address, word_value)`: stores `word_value` in the main memory position pointed by `effective_address`.

The problem with this software cache model is that it does not guarantee cache coherency. Modifications in an SPE cache are not reflected to others SPEs. In this scenario, any arbitrary combination of reads and writes may lead to data inconsistencies. The same inconvenience happens if there are different variables residing in the same cache line due to false sharing.

Now consider the case in which a programmer must access data shared among the SPE threads. The usual method to avoid data race conditions by using mutex locks will also fail since this software cache model does not guarantee data consistency. Therefore, even serializing the access to shared data by using mutexes will not prevent race conditions. In this scenario the only alternative is to explicitly issue a DMA command when accessing shared data inside a critical area, making the source code difficult to read and maintain. The implementation of an STM system on top of the software cache described in the next section solves these drawbacks.

3.2 Transactional Software Cache

We introduce here the TSC model by merging the TL2 transactional memory model with the traditional software cache. Not only this guarantees that each set of memory operations running on an SPE thread are consistent and coherent with other SPE threads, but it also elegantly allows programmers to access shared data without the need of explicit lock operations.

The TSC model does not add more programming complexity than the traditional software cache model does. As explained before, the software cache model does require every data located in main memory to be accessed through the provided macros. However, since DMA operations are hidden from programmers the use of caching is still advantageous. Traditional software transactional memory systems implemented as a library for unmanaged languages also have a very similar characteristic: they require programmers to insert read and write barriers manually.

When we merge both models, we see that both requirements are the same and the only thing that changes is the way the read and write barriers are implemented. From a programmer's point of view, there is no change from the traditional software cache programmability to the TSC programmability.

To illustrate this, we present a practical example of how the source code of an ordered linked-list can be converted into a software cache version and also how this version can be converted to a TSC one.

Figure 1a shows the source code of the `remove` routine of an ordered linked-list and its use in a code block. We want to convert this routine into a software cache version that makes use of the `LOAD` and `STORE` macros to read and write variables from/to main memory. At this time we assume that we will run only one SPE thread accessing the linked-list stored in main memory.

All the bold-face code in Figure 1a are accesses to memory locations that will be in main memory and, therefore, we need to change each read access to use the `LOAD` macro and each write access to the `STORE` macro. For example, the first bold-face code line in figure 1a is an assignment from the variable `prevPtr->nextPtr`. This variable is located in main memory at the address `&(prevPtr->nextPtr)`. Thus we need to change it and add the `LOAD` macro as shown in the first bold-face line of Figure 1b. Figure 1b shows how the list remove routine will look-like after converting it to a software cache model. All the modifications, in bold-face, follow the same style mentioned above.

Now, let us say we want to add more SPE threads executing the same source code in parallel. In this scenario we need to use the TSC model to ensure data coherency. To do so, we must look carefully the source code in the Figure 1a to find out the piece of code that should run atomically. This is our critical section and we should take care of it when converting our source code. A dashed box is around the three lines found as critical section. All we need to do is to add the macros `TM_BEGIN` and `TM_END` guarding this

```
bool_t list_remove (list_t* listPtr, void* dataPtr) {
    list_node_t* prevPtr = list_findPrevious(listPtr, dataPtr);
    list_node_t* nodePtr = prevPtr->nextPtr;
    if ((nodePtr != NULL) &&
        ( compare(nodePtr->dataPtr, dataPtr) == 0 )) {
        prevPtr->nextPtr = nodePtr->nextPtr;
        nodePtr->nextPtr = NULL;
        freeNode(nodePtr);
        listPtr->size--;
        return TRUE;
    }
    return FALSE;
}
int main() {
    ..
    ...
    list_insert(listPtr, value_X);
    list_find(listPtr, value_X);    // Must Always Find X
    list_remove(listPtr, value_X);
    ...
    ..
}
```

(a) Original code

```
bool_t list_remove (list_t* listPtr, void* dataPtr) {
    list_node_t* prevPtr = findPrevious(listPtr, dataPtr);
    list_node_t* nodePtr = LOAD( &(prevPtr->nextPtr) );
    void* node_dataPtr;
    if (nodePtr != NULL) {
        node_dataPtr = LOAD( &(nodePtr->dataPtr) );
    }
    if ((nodePtr != NULL) &&
        ( compare(node_dataPtr, dataPtr) == 0) ) {
        STORE( &(prevPtr->nextPtr), LOAD(&(nodePtr->nextPtr)));
        STORE( &(nodePtr->nextPtr), NULL );
        freeNode(nodePtr);
        long size = LOAD( &(listPtr->size) );
        size--;
        STORE( &(listPtr->size), size );
        return TRUE;
    }
    return FALSE;
}
```

(b) Software Cache code

```
int main() {
    ..
    ...
    TM_BEGIN();
      list_insert(listPtr, value_X);
      list_find(listPtr, value_X);   // Must Always Find X
      list_remove(listPtr, value_X);
    TM_END();
    ...
    ..
}
```

(c) TSC code

Figure 1: Conversion of a linked-list code to TSC

piece of code and we transform the critical area into a transaction as shown in 1c. This is necessary to guarantee that the three list operations guarded by the TM macros are executed atomically. Notice that this is the only modification we need to do, the rest of the code remains the same as the software cache model.

Therefore we can see that the difficulty to transform a conventional source code into a software cache code is the same of transforming the source code into a TSC code. Since this is a mechanical transformation process, it could be made by compilers without programmer intervention.

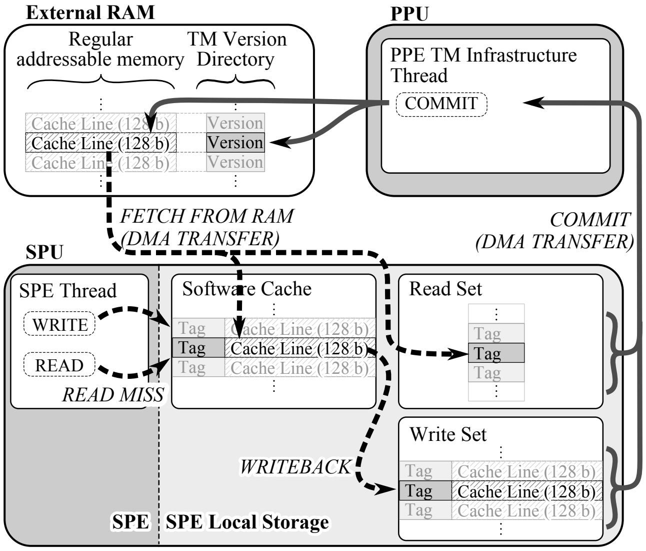


Figure 2: TM Infrastructure on the CellBE

3.3 Infrastructure of TM on CellBE

Our transactional memory model acts as a layer between the software cache in the SPE local store and the accesses to main memory. We leveraged the conventional software cache by adding two new operations: a transactional fetch and a transactional write-back. When a fetch is executed inside a transaction, the transactional version is executed instead of the regular one. The same happens to the write-back routine.

An overview of the infrastructure is depicted in Figure 2. Its main components are comprised of:

- *SPU unit:* contains the read-set, write-set and the software cache structure in SPE local store. There is only one thread running in each SPE processor;
- *PPU unit:* contains the PPE thread responsible to perform the commit procedure;
- *Main memory:* contains the regular addressable memory and the transactional memory directory that keeps track of versioning.

The transaction descriptor, read-set and write-set for each transaction are stored in the SPE local store. Notice that both the read and write sets could also be stored in main memory if a larger set size is required, but we leave this for future work. The transaction descriptor also contains the current transaction version-clock used to validate the reads. The versioning directory is maintained in main memory along with the global version-clock.

When a transaction is initiated it first fetches the current global version-clock from main memory and stores it locally in its transaction descriptor. A transactional fetch inserts the new cache line address into the read-set (if it is not already present). It then verifies whether the write-set contains the fetched address (i.e., the address had been previously modified). In the affirmative case, the address is stored in the cache line array and its current value returned. Otherwise, it is necessary to fetch the current version for the cache line from the TM versioning directory so that validation can take place. If the validation process fails, the transaction is aborted and restarted. Otherwise a new cache line must be fetched from main memory and stored in the cache line array.

A transactional write-back, differently from the conventional write-back, does not transfer the modified cache line to the final address in main memory. Instead, it adds the cache line in its write-set (this is known as lazy versioning). The dashed-line arrows in Figure 2 illustrates the transactional fetch and write-back operations.

Notice that all the aforementioned actions are executed entirely by only one SPE. In order to take advantage of the Cell BE heterogeneous architecture our infrastructure assign some tasks to the PPE processor. When the SPE thread needs help from the PPE to execute an specific task it makes a call to the PPE in a Remote Procedure Call (RPC) fashion. Our RPC-like system was implemented in a similar way as the `spe_callback_handler_register` mechanism, available in the IBM SDK. The RPC message itself is passed as mailboxes. Its arguments may be either passed through mailboxes or DMA depending on their size.

The most important task that the PPE is responsible for is the transaction's commit procedure. When the SPE is ready to commit, it transfers its read and write sets to a predefined temporary main memory location and makes an RPC call to the PPE to request a commit operation. The SPE also sends the current transaction version-clock so as to allow the PPE to validate the transaction. As can be seen, the commit operations is realized by the PPE. The steps involved are similar to the TL2 algorithm: (1) lock the write-set; (2) increment the global version-clock; (3) validate the read-set; (4) make the actual commit, and (5) release the locks.

In the meantime the SPE is waiting for an answer from the PPE to check if the transaction has succeeded or failed. If the commit phase fails, the PPE replies to the SPE with a negative answer and the SPE executes an abort operation restarting the current transaction. If the commit has been succeeded the PPE replies with a positive answer and the SPE continue its normal execution. The PPE also executes other tasks, such as allocate and free memory when requested by one of the SPE cores.

4 Experiments

Our case studies have been tested on a PlayStation 3[1] machine. The PlayStation 3 is provided with a Cell BE processor running at 3.2 GHz with one Power Processor Unit (PPU) and six Synergistic Processor Unit (SPU)[2].

Our evaluation methodology consisted of running each application instance 10 times and taking the average. We divided the measured number of successfully completed transactions by the total execution time to obtain the number of transactions per second and use this number as a metric of comparison.

We ported the Linked-List and the HashTable implementation from the STAMP [3] benchmark to our TSC system on the Cell BE. Using these two data structures, we implemented two micro-benchmarks similar to the IntSet micro-benchmark used in [8]. Additionally, we also ported the Genome application from STAMP [3], as it is a real and big application and can give a more realistic result in the evaluation of our transactional memory system.

All the experiments evaluated were compared against the same application implemented using a global mutex-lock that is acquired when accessing a shared memory region and using the traditional software cache library provided by the IBM SDK.

4.1 IntSet Linked-List

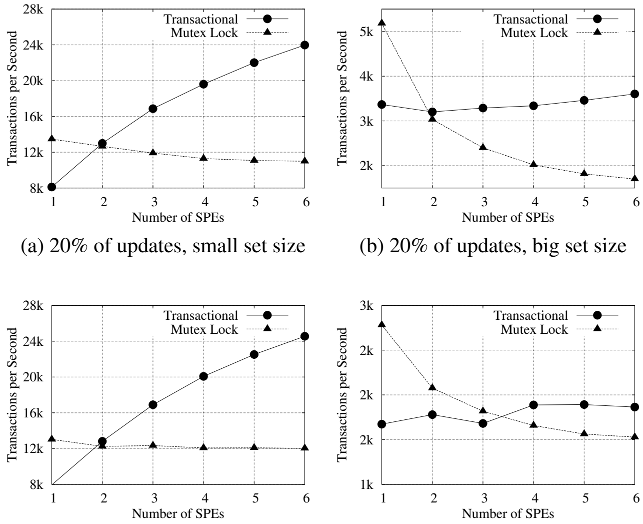
We evaluated the IntSet Linked-List micro-benchmark using two different average set sizes, a small one with 100 nodes and big one with 1000 nodes. We also vary the frequency of updates from 20% to 50% for each set size (half of updates are insert operations and half are remove operations).

Figure 3 shows the number of transactions per second as a function of the number of running SPEs, comparing the mutex-lock implementation with the TSC. Figure 3a shows the IntSet Linked-List using the small set and 20% of updates, Figure 3b uses the big set and 20% of updates, Figure 3c uses the small set and 50% of updates, Figure 3d uses the big set and 50% of updates. These charts show that the performance of the mutex-lock degrades while the TSC increase as the number of SPEs increase.

It is important to note the scalability problem in the liked-list. Figures 3b and 3d show that it is more efficient to use only one SPE with mutex-lock than any other scheme when handling a linked list of big size. One can see that, as the number of concurrent SPEs increases, there is a sharp decrease of performance in the lock version. It is even more

[1] PlayStation is a registered trademark of Sony Computer Entertainment Inc.

[2] From the total of the eight SPUs, one is locked-out on the factory to improve manufacturing yields and another one is reserved for the OS



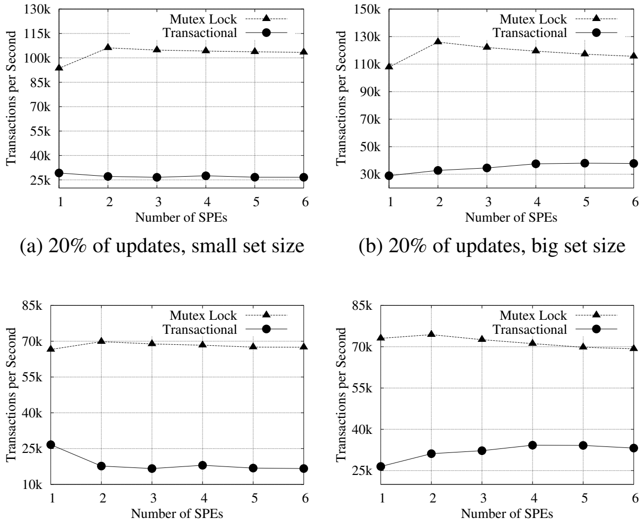
(a) 20% of updates, small set size

(b) 20% of updates, big set size

(c) 50% of updates, small set size

(d) 50% of updates, big set size

Figure 3: IntSet Linked-List



(a) 20% of updates, small set size

(b) 20% of updates, big set size

(c) 50% of updates, small set size

(d) 50% of updates, big set size

Figure 4: IntSet HashTable

critical as the number of nodes in the linked list increases, achieving a slow down of up to 3× compared with 1 SPE.

Additionally, the transactional version has a small improvement of 7% in performance as the number of SPEs increases, but this improvement is not worth compared to the mutex-lock version with one SPE. In the ordered linked-list, as its size grows up, the probability that one transaction will abort increases proportionally because any list operation will traverse most part of its nodes to reach the desired searched node. Due to the increased number of aborts, the transactional version does not scale better in a big-sized ordered linked-list.

4.2 IntSet HashTable

We evaluated the IntSet HashTable micro-benchmark also using two different average set sizes, a small one with 100 nodes and big one with 10000 nodes. The frequency of updates are the same as of the Linked-List as cited before. The charts of transactions per second as a function of the number of running SPEs are plotted in Figure 4.

Figure 4 shows that the mutex lock version is faster when using the IntSet HashTable benchmark, but this can be explained. This micro-benchmark does one hashtable operation per transaction inside a for loop and there is no other useful work besides this. The hashtable operation is almost always small and fast, thus, when running the transactional version, the total time of the benchmark is dominated by the overhead of the transaction. Additionally, we can note that the mutex-lock version degrades the performance as the number of SPEs running increase in contrast to the transactional version that has its performance increased.

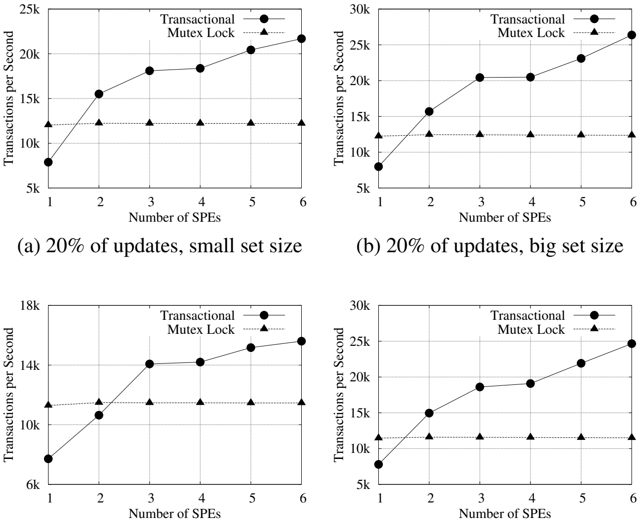
We assume that real applications will use larger transactions than just one single hashtable operation. This assumption is corroborated in the next session when we evaluate the Genome application. To simulate this scenario with the IntSet HashTable benchmark, we increased 10× the transaction duration as if the application was executing more work. For this modification, we plotted the same charts as the original IntSet HashTable. These charts are shown in Figure 5. Notice that, in this case, the TSC had the performance improvement while the mutex lock presented almost the same behavior.

To analyze the impact of the transaction size in the HashTable IntSet benchmark we plotted the chart in Figure 6 using 6 SPEs. It shows that increasing the amount of work executed inside a transaction, the performance of mutex-lock degrades drastically while the TSC continues to perform well, degrading in a near-linear manner.

4.3 Genome: A Real Application

Genome is a gene sequencing multi-threaded application. Its input, a set of gene-segments, and output, a set of sequenced segments, are shared by all threads. It makes use of the linked-list and the hashtable with a mix of small, big transactions and also non-transactional code.

We evaluated the Genome with two different sizes of input. The small size has the following parameters: *Length of gene: 500*, *Length of segment: 16*, *Number of segments: 2000*. The big size has the following parameters: *Length of gene: 4000*, *Length of segment: 16*, *Number of segments:*



(a) 20% of updates, small set size

(b) 20% of updates, big set size

(c) 50% of updates, small set size

(d) 50% of updates, big set size

Figure 5: IntSet HashTable modified

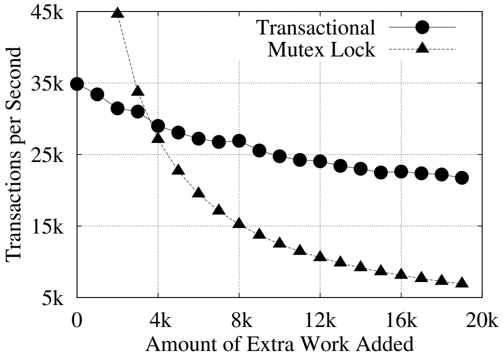
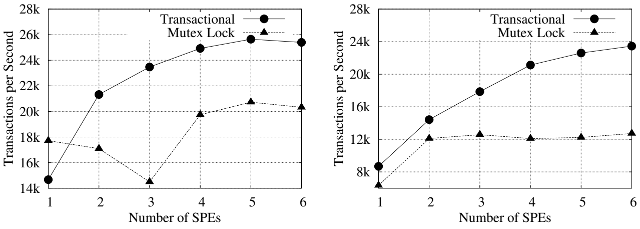


Figure 6: Analysis of transaction size

500000. The charts plotted in Figure 7 shows the performance of Genome application for these two configurations.

As we were expecting, the Genome application performed better with the TSC than with mutex-lock, showing that our assumption that a real application performs more work inside a transaction than just one hashtable operation is true.

As the charts show, the Genome application using our TSC approach obtained a performance gain of 84% with a big input set using 6 SPEs and a gain of 24% with a small input set using 5 SPEs, compared to the lock version. Additionally, our approach scale much better when increasing the number of SPEs. In the big input size, the TSC scaled from 8683.42 transactions per second (tps) using 1 SPE to to 23451.04 tps (2.7×) using 6 SPE while the mutex lock scaled from 6346.37 tps to 12723.48 tps (2.0×).



(a) Small input size

(b) Big input size

Figure 7: Genome application

The chart of the Genome application with small input size, presented in Figure 7a, has an anomaly that can be seen in the mutex-lock performance. This chart shows that the mutex-lock version decreases the performance rapidly when running with 3 SPEs and increase its performance again when running with 4 to 6 SPEs. We suspect that this anomaly is caused by some odd synchronization behavior of the Genome's algorithm implementation. We believe that this anomaly is amortized or hidden as the input size increases, because this anomaly is not present in the big input size.

5 Related Work

A number of programming models can be used for the Cell BE processor, ranging from common parallel programming models (i.e. pipelining, streaming) to Cell BE specific programming models [9] (i.e. SPU-let, function offload, device extension, computational acceleration, asymmetric thread runtime). For a discussion about related work we are particularly interested in the shared-memory programming model. It is important to notice that programming models can be mixed and adapted according to the developer needs.

As have been pointed out before, shared-memory programming on the Cell is feasible [13]. Although it is necessary to perform DMA operations in order for the SPE cores to access system memory, the access latency is in the same order of magnitude as the latency of a miss in the L2 cache on conventional architectures.

One of the first abstractions targeting shared memory programming is that of software cache. One such implementation is currently distributed with the IBM SDK. The IBM's XL compiler [5] also provides transparent support for shared-memory programming through compiler-controlled software cache and partitioning of code and data. Software cache also appears in the work of Balart et al. [2]. However, the software cache in their approach is used to decouple computation from communication so that the two dimensions can be overlapped for better performance. In that aspect, their technique is orthogonal to ours. Also, none of the cited works allows SPE code to access shared memory concurrently, that is, they lack consistency. There has also been an attempt to implement OpenMP for the Cell architecture as reported in [12].

A lot of research has been conducted to explore the implementation space for software transactional memory [7, 10, 11]. In the context of this paper we pay particular attention to time-based unmanaged STM systems operating at the word-level granularity. Our implementation of STM for the Cell architecture relies on the Transactional Locking II (TL2) algorithm [4]. A similar implementation is also employed by tinySTM [6]. They differ on some relatively subtle aspects. For instance, tinySTM employs encounter-time locking whereas TL2 performs locking only at commit time. An important feature of this class of STM implementation is that it guarantees that values read by transactions are always consistent.

The main difference from our implementation to that of TL2 or tinySTM is that we had to deal explicitly with communication among the cores through DMA. Also, the architecture asymmetry required special attention to some aspects that are simply transparent in conventional architectures. For instance, the limited amount of local memory in each SPE (256K) and the implementation of a cache system in pure software.

6 Conclusions

This work presented a software transactional memory implementation for an asymmetric architecture, namely the Cell BE processor.

We implemented and evaluated our TSC model using a micro-benchmark and also a real-world application. Results showed a performance gain of 84% with a large input set and a gain of 24% with the small one for the Genome application when compared to a lock-based implementation. Additionally, our approach scale much better with the increasing number of SPE cores. We conclude that our approach is efficient for the benchmarks and the real application evaluated.

One can also argue that, since all the data transfers between the SPE and the main memory are controlled by the TSC system, it becomes feasible to control the memory traffic to allow strong isolation. We intend to formalize and explore this concept in future work. Finally, integrating our system into a compiler framework will make it possible for programmers to explore STM on the Cell BE transparently.

7 Acknowledgements

This work was partially supported by CNPq, FAPESP and IBM.

References

[1] IBM SDK for multicore acceleration: Example library API reference, version 3.0, 2007.

[2] J. Balart, M. Gonzalez, X. Martorell, E. Ayguade, Z. Sura, T. Chen, T. Zhang, and K. O'brien. A novel asynchronous software cache implementation for the cell-be processor. *Proc. of the LCPC 07*, 2007.

[3] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th ISCA*. 2007.

[4] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proceedings of the 20th DISC*, 2006.

[5] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing compiler for the cell processor. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, 2005.

[6] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th PPoPP*, pages 237–246, 2008.

[7] T. Harris, A. Cristal, O. Unsal, E. Ayguade, F. Gagliardi, B. Smith, and M. Valero. Transactional memory: An overview. *IEEE Micro*, 27(3):8–29, 2007.

[8] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer. Software transactional memory for dynamic-sized data structures. In *Twenty-Second Annual ACM SIGACT-SIGOPS PODC*, 2003.

[9] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[10] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[11] A. McDonald, B. D. Carlstrom, J. Chung, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Transactional memory: The hardware-software interface. *IEEE Micro*, 27(1):67–76, 2007.

[12] K. O'Brien, K. O'Brie, Z. Sura, T. Chen, and T. Zhang. Support OpenMP on Cell. In *International Workshop on OpenMP (2007)*, 2007.

[13] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. MPI microtask for programming the Cell Broadband Engine processor. *IBM Systems Journal*, 45(1):85–102, 2006.

[14] H. Sutter and J. R. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.