A Transactional Runtime System for the Cell/BE Architecture

A. Baldassin[a,*], F. Goldstein[b], R. Azevedo[b]

[a] *UNESP – Univ Estadual Paulista, DEMAC. Av. 24A, 1515 - Bairro Bela Vista, Rio Claro - Brazil*

[b] *University of Campinas (UNICAMP), IC. Av. Albert Einstein, 1251 - Cidade Universitaria, Campinas - Brazil*

Abstract

Single-core architectures have hit the end of the road and industry and academia are currently exploiting new multicore design alternatives. In special, heterogeneous multicore architectures have attracted a lot of attention but developing applications for such architectures is not an easy task due to the lack of appropriate tools and programming models. We present the design of a runtime system for the Cell/BE architecture that works with memory transactions. Transactional programs are automatically instrumented by the compiler, shortening development time and avoiding synchronization mistakes usually present in lock-based approaches (such as deadlock). Experimental results conducted with a prototype implementation and the STAMP benchmark show good scalability for applications with moderate to low contention levels, and whose transactions are not too small. For those cases in which a small performance loss is admissible, we believe that the ease of

[*] Principal corresponding author

Email addresses: `alex@rc.unesp.br` (A. Baldassin), `felipe.goldstein@students.ic.unicamp.br` (F. Goldstein), `rodolfo@ic.unicamp.br` (R. Azevedo)

programming provided by transactions greatly pays off.

Keywords:
Multiprocessors, Parallel Programming, Transactional Memory

1. Introduction

Extreme power dissipation and microarchitectural limitations have made the semiconductor industry bet its future on multicore architectures, or chip multiprocessors as they are usually called [1]. The general trend is to replicate a microprocessor core and integrate them into a single chip, probably lowering the clock rate to address the power dissipation issue.

Although the vast majority of current multiprocessors employ an homogeneous approach, wherein only one type of core is used, new designs are starting to exploit heterogeneity. These asymmetric chip multiprocessors are usually comprised of a general purpose core along with simpler and more specialized cores. The reasoning behind heterogeneous architectures is that they are able to address a larger class of applications more efficiently in terms of throughput and energy per instruction [2].

The main impediment for a wider adoption of asymmetric architectures lies in the appropriate programming models and tools for software development. Since different cores coexist in the same system, the lack of proper abstractions requires programmers to know a large fraction of the underlying microarchitecture to exploit performance to the fullest, lengthening the software production cycle.

Take as an example the Cell Broadband Engine (Cell BE) architecture designed by Sony, Toshiba, and IBM [3, 4]. A Cell BE is comprised of a multithreaded PowerPC (PPE) and eight specialized cores called synergistic processor elements (SPEs). Each SPE has its own local memory, segregated from the global memory seen by the PowerPC. Moving data between global and local memories requires programmers to issue specific direct memory access (DMA) commands. Coherence between these memories must also be handled by programmers since it is not enforced by hardware. This significantly complicates shared memory concurrent programming, the most common concurrent programming model, on Cell. Moreover, PPE and SPEs have two different instruction set architectures and therefore two distinct sets of binary tools are required.

We propose a transactional runtime system in order to simplify the development of shared memory concurrent software for the Cell BE architecture. The main abstraction of our system is that of atomic blocks (or transactions), an area of extensive research but whose implementation focus is mostly on homogeneous architectures [5, 6]. The contributions of this article are as follows:

- We describe a transactional programming model for the Cell/BE architecture: programmers mark shared variables with a specific attribute and confine the code that touches them into atomic blocks.
- We present a software implementation of a transactional runtime system for the Cell BE architecture based on the state-of-the-art Transactional Locking II (TL2) algorithm designed by Dice et al. [7]. The main modification of the algorithm regards the commit operation which is realized in two stages, the first performed by a synergistic processor and the second by the main processor.

- We conduct a thorough evaluation of our runtime system using the STAMP benchmark suite [8]. We show how the system behaves and conclude that the best case scenario is for those applications in which transactions are not too small and contention level is moderate to low.

The rest of this article is organized as follows. Section 2 gives a background on the Cell BE architecture and memory transactions, also describing related work. Section 3 presents the programming model and how it simplifies software development for Cell BE. We discuss design choices and present an implementation of the runtime system in Section 4. Section 5 shows a comprehensive performance analysis and the achieved results, and we conclude the article in Section 6.

2. Background

In this section we briefly present the background necessary to the subsequent sections (Cell BE architecture and memory transactions) and discuss related work.

2.1. Cell BE architecture in a nutshell

The Cell BE architecture is comprised of nine processing elements, a Memory Interface Controller (MIC), a Broadband Engine Interface (BEI), and a high performance coherent bus, the Element Interconnect Bus (EIB). The EIB connects the processing elements, the MIC, and the BEI together. A diagram with the main components is shown in Figure 1.

The processing elements can be split into two different categories: one PowerPC processor (PPE) and eight synergistic processors (SPEs). The PPE

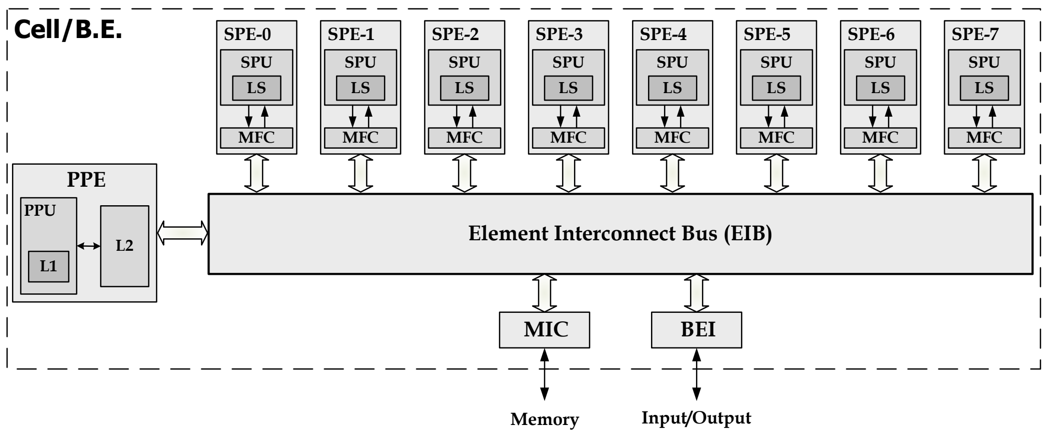

Figure 1: Cell BE Architecture.

is comprised of a 64-bit execution core based on the PowerPC architecture (with two simultaneous threads) and an L1 cache, typically known as PPU (PowerPC Processor Unit). An external L2 cache is also part of the PPE. The main purpose of the PPE is to execute the operating system and manage the synergistic elements. The SPEs are SIMD processors specialized in data-intensive processing. Each one has a simple execution core (without out-of-order execution and branch prediction), a 128 128-bit register file, and a 256KB local storage (LS), together referred as SPU (Synergistic Processor Unit). Communication with main memory, PPE, and other SPEs is handled by the Memory Flow Controller (MFC).

There are two main differences between the PPE and SPE. Firstly, the corresponding instruction set architectures are incompatible, leading to two distinct sets of compilers and binary tools. Secondly, whereas the PPE accesses global memory by means of simple load/store instructions, an SPE needs to issue a DMA transfer through its MFC, since its load/store instructions affect only the local storage. Besides DMA operations, the MFC also allows communication among SPEs and PPE through a *signal mechanism* and *mailbox*.

2.2. Transactions at a glance

A transaction is a sequence of instructions that is either entirely executed, meaning the changes performed by the instructions logically take place at the same point in time, or not executed at all, meaning no partial result is left. In the former case, we say that the transaction committed, whereas in the latter case the transaction aborted. It is a common practice to restart a transaction once it is aborted.

Memory transactions raise the abstraction level and make it simpler to deal with synchronization code in a concurrent application. While a lock-based scheme requires programmers to come up with a locking protocol and decide among different locking granularities (ease of programming and performance tradeoff), transactions move most of the burden from programmers to the implementation system.

Transactional memory (TM) is an active area of research and implementations have been devised both in hardware (HTM) and software (STM) (hybrid approaches also exist) [5, 6]. The transactional runtime system proposed in this article falls into the STM category and, therefore, we focus our discussion on this category of implementations. An Application Programming Interface (API) for STM is usually comprised of the following primitives:

- `begin_atomic` – creates a checkpoint and starts a transaction;

- `end_atomic` – ends the transaction and attempts to make the changes permanent (the transaction is automatically rolled back otherwise);
- `stm_load` and `stm_store` – read and write barriers, respectively. These barriers must be used instead of conventional load and store instructions when accessing shared data.

In general, there are two broad classes regarding the design space of STM: non-blocking and blocking. Early software implementations (such as DSTM [9] and RSTM [10]) were non-blocking, avoiding the use of locks in their design. However, most non-blocking designs at the time incurred high overhead due to indirection and validation, which prompted researchers to investigate blocking implementations. One of the most famous blocking algorithms is the Transactional Locking II (TL2), designed by Dice et al. [7]. Since our approach is based on TL2, we present in the following paragraphs a general overview of this algorithm and how it compares to other lock-based implementations. More details are described in Section 4, when we present our implementation.

The TL2 algorithm relies on two main shared data structures: a *Global Clock* (GC) and an *Ownership Record Table* (ORT). The GC can be seen as a timestamp and is used to maintain system-wide consistency. The ORT stores a *versioned lock* for each shared memory address. A hash function is used to map these addresses into a versioned lock in a per-stripe fashion. Each versioned lock serves two purposes: if the corresponding memory address is not locked (indicated by the least significant bit) it holds the corresponding version number based on the GC; otherwise a transaction has locked the respective address, preventing access from other transactions.

Briefly, the operation of each transactional primitive in TL2 is as follows. Upon starting (`begin_atomic`), each transaction retains a local copy of the GC. A read barrier (`stm_load`) makes sure the execution is operating on a consistent state by checking the transaction local version number against the versioned lock of the loaded word. A write barrier (`stm_store`) causes the referenced address and corresponding value to be stored locally in a write buffer. The commit phase (`end_atomic`) starts by acquiring the locks for the shared addresses contained in the write buffer and proceeds to increment the GC atomically. After that, the read set is validated, the changes are committed into main memory and the locks are released. If inconsistency is found in any of the steps, the transaction is aborted and re-executed.

The main advantage of TL2 over their lock-based counterparts (such as McRT-STM [11] and Bartok-STM [12]) is that it provides opacity [13] (memory view is always consistent) without requiring incremental validation. Recent lock-based implementations such as TinySTM [14] and SwissTM [15] also use the timestamp technique and the basic metadata used by TL2. They mostly differ on how data versioning and conflict detection are performed. Whereas TL2 uses optimistic concurrency (conflicts are detected at commit time) and deferred updates (tentative writes are stored locally), TinySTM provides pessimistic concurrency (conflicts are detected eagerly) and can also make use of direct updates (tentative writes are stored in main memory and old values locally). SwissTM uses optimistic concurrency for read/write conflicts and pessimistic concurrency for write/write conflicts.

2.3. Related Work

One way to improve programmability on the Cell architecture is to use a technique known as *Software-Managed Cache* (SMC), which effectively simulates a cache using each SPE local store. Therefore, programmers are freed from the burden of manually dealing with DMA transfers and, at the same time, may benefit from performance gains due to the temporal and spatial locality provided by the software cache. Indeed, SMCs are a common technique used by Cell applications and one such implementation is provided with the Cell SDK library [16].

More recently, researchers have proposed new and more advanced implementations of SMCs for the Cell BE processor [17, 18, 19, 20], including prefetching of irregular references [21]. Although SMCs guarantee that an SPE local store is coherent with the external main memory, they do not provide coherence among the SPE local stores. Therefore, programmers still need to handle explicit data transfers if more than one SPE is running code that shares data. Our approach provides the benefits of the transactional model for the Cell architecture, effectively hiding the complexity of handling data transfers just like SMCs. Different from SMCs, however, our runtime system guarantees that all accesses to the SPE local stores and global memory are coherent.

The closest work to ours is COMIC [22]. COMIC API provides a typical shared memory programming environment in which multiple threads can be spawned. A write to a shared location issued by a SPE thread is visible to reads from other SPE threads. Similar to our work, COMIC uses a centralized approach wherein synchronization services are provided to the SPEs by the PPE. The main difference between our work and COMIC is the programming model adopted: while COMIC provides lock-based synchronization, we take a leap ahead and allow Cell programmers to experiment the benefits of the transactional model.

There is a large body of work on software transactional memory implementations for homogeneous processors [9, 10, 11, 12**?** , 14, 15, 23, 24, 25]. While some investigation have been carried out on adapting the transactional model for clusters [26, 27], the research on evaluating STM designs on heterogeneous architectures is very limited. Lee et al. [28] present such an approach but there is not enough information to compare their system to ours. Also, no experimental results are reported. In this article we provide a transactional system implementation and evaluation for an heterogeneous system, the Cell/BE architecture.

3. Programming Model

Shared memory programming on Cell is typically achieved by creating a PPE main thread and launching one or more SPE worker threads from this main thread. The standard way to perform synchronization among threads is through locks and condition variables, whose drawbacks are well-known [29]. The atomic operations necessary to implement the synchronization mechanism are provided by the *load-linked* and *store-conditional* instruction pair for the PPE. For the SPE, the MFC provides a similar semantics but requires DMA operations and, therefore, extra overhead is added.

Our proposed programming model uses a different scheme for synchronization. Instead of using lock-based primitives, we provide a transactional model. Programmers are responsible to group the instructions that are required to be executed indivisibly into an atomic block. Furthermore, and contrary to typical STM libraries, programmers use the `__ea` qualifier to mark shared variables so that the compiler can insert the transactional read and write barriers automatically.

For the sake of discussion, consider a simple scenario: each SPE must increment a global counter in the PPE address space. A possible implementation using a conventional STM library is shown in Figure 2a. Notice that the value of *counter* (line 1) must be correctly initialized before the code is actually executed (the SPE could receive the pointer value from the PPE at initialization, for example). The programmer then uses the calls `begin_atomic()` and `end_atomic()` (lines 3 and 7, respectively) to specify the atomic block, that is, the sequence of instructions that must be executed atomically. In a typical STM library, the programmer must also explicitly instrument the code by inserting memory barriers to read from (line 4) and write to (line 6) shared memory.

During development, correctly performing manual instrumentation is both cumbersome and error-prone. Therefore, we provide a system in which the compiler automatically inserts the transactional barriers. All that is required from programmers is to add the address space identifier `__ea` to the pointer declaration. Figure 2b shows how the same program of Figure 2a is coded in our system. Notice the addition of the `__ea` qualifier in line 1 and the removal of the instrumentation code, resulting in a much simpler implementation.

As the aforementioned example has shown, having the compiler to instrument the code is a key feature of our system. Not only does it allow

```
int *counter; // must be initialized with PPE address

begin_atomic()
  int value = stm_load(&counter);
  value++;
  stm_store(&counter, value);
end_atomic()
```

(a) Explicitly using an STM library

```
int __ea *counter; // must be initialized with PPE address

begin_atomic()
  *counter++;
end_atomic()
```

(b) Compiler support

Figure 2: Incrementing a global counter.

programmers to quickly develop applications, but it also avoids the pitfalls in manually dealing with the transactional barriers. The extra cost of adding the `__ea` qualifier is completely justified and, moreover, it also helps in documenting the code. As an example, we measured the number of extra lines needed to code the genome application from the STAMP benchmark [?], with and without compiler support. Compared to the original version, the code size (in number of lines) with compiler support increased only by 2%, whereas a 29% increase was seen in the case of manual instrumentation.

4. Runtime Implementation

The implementation described here makes use of the SPU Runtime Library Extensions [?]. In fact, the `__ea` address space qualifier language was first introduced as an extension to the SPU library and implemented in the GNU Compiler Collection (GCC) with the purpose of facilitating data sharing between an SPE and the PPE. Whenever the SPU GCC compiler finds a reference to a pointer variable qualified as `__ea`, it appropriately generates a call to the SMC library distributed with the IBM SDK. The SMC API is primarily comprised of two functions:

```
void *__cache_fetch(__ea void *effective_address)
```

the variable at the given effective address is first looked up in the software cache and, in case it is present, returned. Otherwise, a DMA operation is performed to bring the data from system memory to the software cache. This function also takes care of cache replacement operations such as writing back dirty cache lines.

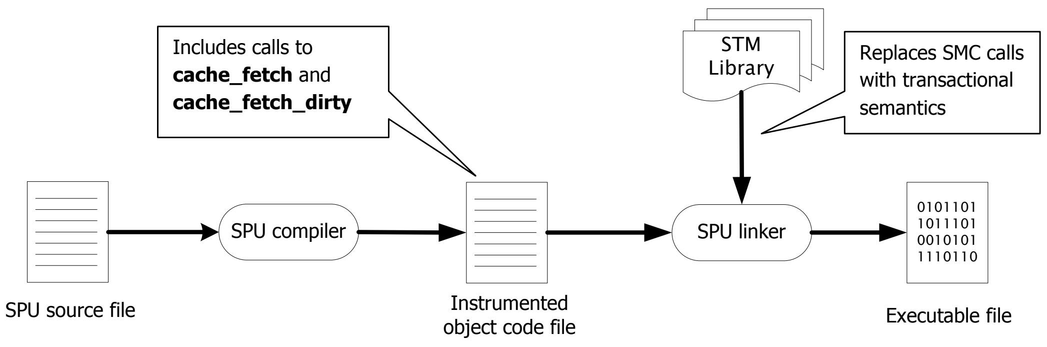

Figure 3: Generation flow.

`void *__cache_fetch_dirty(__ea void *effective_address, int nbd)`

> same behavior as `__cache_fetch`, but the given `nbd` number of bytes are marked as dirty in the cache. Therefore, this call is used in case of store operations.

The reader should notice the similarity between these functions and the transactional read and write barriers. Our implementation leverages the SMC library so that programmers can also use transactions. Instead of using the SMC library, we instruct the linker to use our STM library. The read and write barriers are implemented as wrappers to the `__cache_fetch` and `__cache_fetch_dirty` functions, respectively. Figure 3 shows an overview of the generation process. The SPU compiler receives as input the application source file written with the transactional model. It then generates an instrumented object code containing calls to both the STM and SMC libraries (in case of the `__ea` qualifier). However, the SPU linker only references the STM library when generating the executable file, since the corresponding SMC functions are mapped to the transactional read and write barriers.

In the rest of this section we describe the design of our STM system and discuss the implementation of the main transactional primitives (start, commit, read and write barriers) for the Cell/BE.

4.1. Design Choices

A crucial aspect to consider when devising an STM system for an heterogeneous architecture is which algorithm to use. As discussed in Section 2.2, STM designs can be broadly classified as either non-blocking or blocking. We ruled out non-blocking algorithms for two main reasons: (i) high cost of validation, and (ii) most of the designs are intended for object-oriented languages, whereas our primary interest is on procedural languages, most notably the C language.

A natural choice among the blocking options is the TL2 algorithm [7]. We also considered alternatives such as TinySTM [14] and SwissTM [15], but ended up selecting TL2 as basis for our implementation due to the reasons discussed in the following. Firstly notice that, since heterogeneous architectures have segregated memories, it is of paramount importance to reduce the number of costly DMA transfers among local and shared memories. Therefore, transactional primitives that are most often used, specially read and write barriers, must avoid excessive communication with external memory.

The original TL2 algorithm needs three shared memory accesses for the read barrier and none for the write barrier. Since TinySTM uses pessimistic concurrency, it requires an atomic read-modify-write operation inside the write barrier, which is very expensive for the SPE to perform. In addition to requiring an expensive memory operation for the write barrier, SwissTM also demands extra memory reads (usually four) for the read barrier, since it manipulates two versioned locks in the ORT to allow for a mix of pessimistic (for write/write conflicts) and optimistic concurrency (for read/write conflicts), and both need to be accessed during a transactional read. This restriction can be removed by organizing the ORT such that the two versioned locks are contiguous in memory, allowing a single DMA operation to bring both locks into the SPE local memory. Note, however, that the extra costly memory operation in the write barrier cannot be avoided.

Another source of overhead common to TinySTM and SwissTM emerges when a transaction needs to abort due to inconsistent reads. Since these designs employ an encounter-time locking approach, all locked words must be released upon an abort. This strategy would add excessive overhead to the abort operation on an heterogeneous architecture, and thus we refrained from using it. On the other hand, TL2 only performs locking at commit time and the cost to abort a transaction is almost non-existent.

The most important design decision we made was to avoid having the SPE issuing a large quantity of DMA operations. Therefore, as the previous discussion suggests, our algorithm is based on TL2. TinySTM and SwissTM do have a performance advantage when aborts are rare, since they do not need to acquire the locks for the write set during commit. However, this strategy sounds far less appealing for the Cell/BE architecture due to the prohibitive price of performing a shared atomic operation inside the SPE. In order to avoid excessive SPE overhead, our design splits the commit operation between the SPE and PPE so that costly operations such as read set validation and locking are performed entirely by the PPE (more details are presented in Section 4.5).

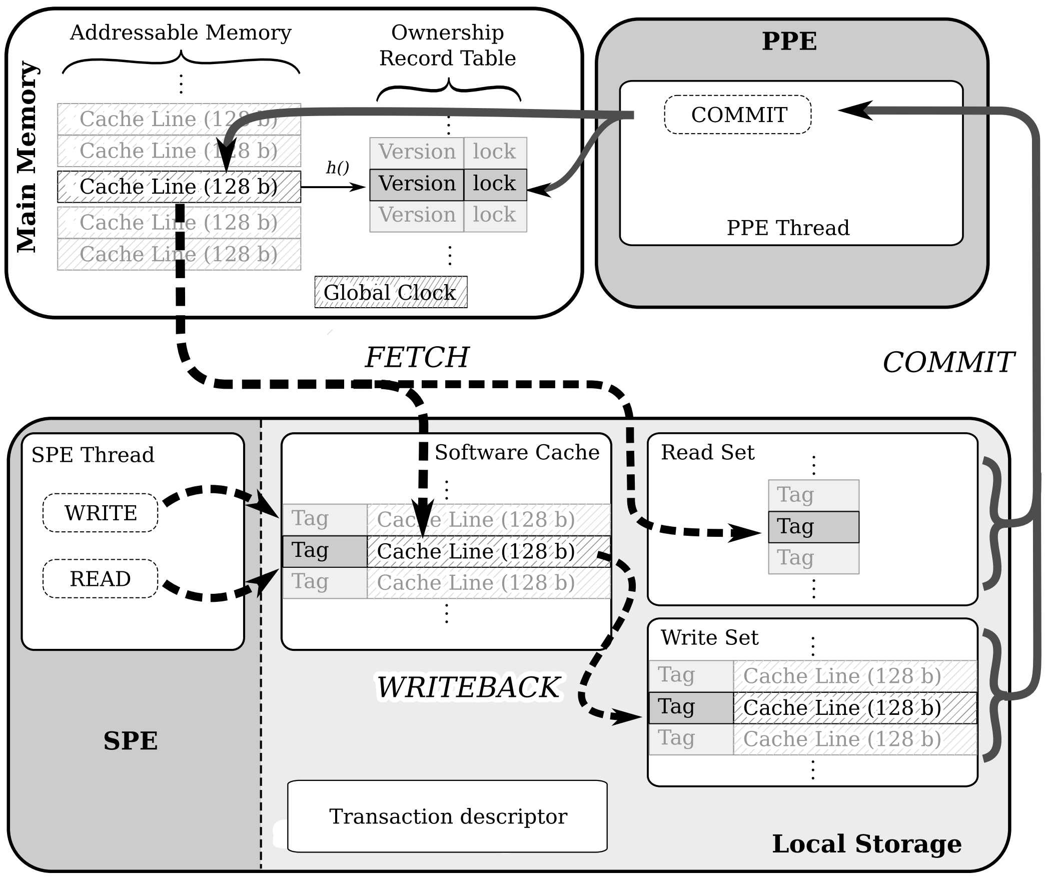

Figure 4: Implementation infrastructure.

4.2. STM Library Infrastructure

For the sake of presentation, we opt to give a informal description of the functionality of our STM system. We make use of Figure 4 throughout the rest of this section to present the main operations and how they were implemented. The following components, functionality, and metadata are part of the system:

Main memory: The main structure kept in shared memory is the ownership record table (ORT). The ORT itself can be seen as a tagless hash table and each of its records contains a version number and a lock bit. Every addressable memory position is mapped into the ORT by means of a hash function. Notice that two or more different addresses can be mapped to the same table record and may cause false aborts: a situation wherein a transaction is aborted when it could possibly proceed.

Since our implementation works at the granularity of the SMC line size, all addresses contained in the same cache line are mapped to the same ORT entry. We use a simple hash function to accomplish this:

$$h(a) = (a >> \log_2 cachelinesize) \bmod ORTsize$$

where a is the referenced address. Function $h(a)$ returns the ORT index of address a. A larger line size favors spatial locality but can also induce false sharing which, in turn, might lead to spurious aborts. We conduct a detailed analysis of this issue in Section 5.1.

A global clock (GC) is also stored in main memory and it is the central mechanism used by TL2 to maintain consistency. The GC is read every time a transaction starts and it is atomically incremented during the commit stage.

PPE: For every transaction running in an SPE thread there is a peer thread hosted in the PPE. The main purpose of this thread is to perform the commit operation, as described in more detail in Section 4.5. An SPE can also offload certain jobs (such as global memory allocation) to the PPE thread by using an RPC-like mechanism.

SPE: Besides the executing thread, every SPE also maintains a set of meta-data information in its corresponding local storage, including the SMC, the read and write sets, and the transaction descriptor. Basically, the transaction descriptor keeps the transaction status (active, committed or aborted), a pointer to the ORT, and a version number. The SMC implementation is based on the one provided with the IBM SDK 3.1 [16].

We now discuss how the main transactional primitives are implemented. The primitives to start a transaction and the read and write barriers follow very closely the original TL2 algorithm. We particularly specialized the commit operation to take into account the heterogeneous nature of the Cell/BE architecture.

4.3. Starting a Transaction

When a transaction starts (call to `begin_atomic`), a copy of the GC is fetched from main memory and stored in the version field of the transaction descriptor. The read and write sets are cleared, the SMC contents are flushed, and the transaction status is set to active. During its lifetime, a transaction will call the read and write barriers to access shared memory. When a call to `end_atomic` is found, the STM system will attempt to commit the written data into main memory. An important aspect of every application is the need to deal with dynamic memory. Our STM system provides a transactional version of the standard `malloc` and `free` functions. The allocation and deallocation processes are actually controlled by the PPE thread.

4.4. Transactional Reads and Writes

Before discussing the actual reading process, we need to describe how a read value is validated. Validation is crucial to the correct execution because it guarantees that the states accessed by a transaction are consistent. To validate a given effective address, its corresponding record (version number and lock bit) must be first retrieved from the ORT. Therefore, a DMA operation is necessary if validation must be performed by an SPE. A variable read by a transaction is valid if the corresponding lock bit is not set and the version field is less than the transaction version number. This certifies that no other transaction is writing to the same position, and that no transaction has written the same variable since the current transaction started. If the check fails, the transaction is aborted and restarted.

A call to `stm_load` takes as an argument the effective address and returns the corresponding data. Programmers always deal with at most 64-bit values (the SPU standard data size), but internally a whole cache line is stored in the SMC for performance reasons. If the required data is already in the cache a hit occurs and the value is immediately returned. If the data is not present, it is looked up in the write set first. If it is still not found, then the data must be fetched from main memory. However, before retrieving the actual data, the transaction must be validated as explained previously. The TL2 algorithm requires another validation step after the data is read. Once the transaction is validated, the read set is updated (only the tag field is stored in the read set – see the FETCH label in Figure 4). From the SPE point of view, the read barrier is expensive since it might require three ordered DMA accesses: two for validation and one for the actual data.

The implementation of the transactional write barrier (function `stm_store`) is much simpler than the read barrier because it does not need to perform any kind of validation. A write operation takes as an argument the effective address and a value to be stored. Instead of writing the value directly into the transaction write set, we simply store the value in the SMC. Only when an SMC dirty line must be replaced the write set is actually used. To implement this functionality we modified the writeback subroutine of the SMC to store the cache line into the write set instead of main memory (see the WRITEBACK label in Figure 4).

4.5. Committing a Transaction

The commit operation is the longest and most complicated one. First of all, it is split into two parts in our system: a first phase executed by an SPE, and a second one performed by the peer PPE thread. This is the main aspect differing our approach from the original algorithm.

When a call to `end_atomic` is found, the SPE first flushes the contents of the SMC. This guarantees that all dirty cache lines are moved to the transaction write set. The SPE then initiates a sequence of DMA requests to transfer the contents of the read set, the write set, and the transaction version number to a specific location in main memory (this location is set by both PPE and SPE during system initialization). After the transfers are completed, the SPE sends a commit message to the PPE (see the COMMIT label in Figure 4). While the PPE is processing the second phase, the SPE waits for a message with the commit status: either the transaction succeeded or failed, in which case the transaction is restarted.

When the PPE receives the commit message, it can immediately start the second phase since the data required by the operation (read and write sets, plus the transaction version number) were already transferred by the SPE. The first step performed by the PPE thread is to acquire the lock for every address in the write set. This is accomplished by using an atomic operation to set the lock bit of the corresponding record in the ORT. If the atomic operation fails, the commit procedure ends and the SPE is notified. After locking the entire write set, the PPE increments the GC. In the next step, every address in the read set must be validated. In case of any inconsistency, the lock bits acquired in the first step must be released and a fail message is returned to the SPE. When the read set is completely validated, the transaction is said to be logically committed. It remains to move the changed data to their actual place in main memory and update their corresponding record in the ORT: new GC becomes the version field, and the lock bit is unset. A success message is then returned to the SPE.

4.6. Aborting a Transaction

Since our algorithm performs commit-time locking and deferred updates, the cost involved in aborting a transaction is almost non-existent. When a conflict is detected and the transaction must be aborted, the SPE simply restarts the transaction by performing a long jump to the `begin_atomic` operation. We do not perform any kind of backoff before restarting, since they did not provide any performance benefits during our tests. Notice that a transaction needs to be aborted in two situations: (1) during a inconsistent read operation in the read barrier; (2) during the commit operation, if the write set could not be locked or the read set validation failed.

4.7. Limitations

In its current state our design does not support nested transactions, although we believe that the flattening model (a transaction and its nested transactions are treated as one large transaction) could be easily integrated into the design. Another limitation regards the size of the transactional read and write sets. Since they are kept in the SPE local storage, there is the possibility of overflows to happen. At the present time we simply raise an exception and abort the program. This problem can be circumvented by using space on external memory when an overflow is detected, but we leave this task as future work.

We would also like to note that our current prototype implementation is not fine-tuned for performance on the SPE, and should be seen as a proof-of-concept of the feasibility of the transactional model on the Cell/BE architecture. We make use of a basic form of branch hinting since the SPE penalty for misprediction is high (18 cycles), but no simdization optimization is performed. Therefore, there is plenty of room left for improvement. We also intend to investigate the performance impact of optimizing compilers such as the one presented in [30] as future work.

5. Experimental Results

In this section we present a thorough evaluation of a prototype implementation of our proposed transactional runtime system. For the experiments conducted in this section we make use of a 3.2GHz Playstation 3 (PS3) machine running Red Hat Linux (kernel 2.6.24). We report results for 6 SPEs, since 2 out of the 8 are not available on PS3 consoles. All the results reported in this section are averaged over 30 executions and presented with a 95% confidence interval.

The applications used in the experiments are compiled with the GNU compiler toolchain available in the IBM Cell SDK 3.1 (GCC version 4.1.1). For the experiments that required SPE code instrumentation (Sections 5.1 and 5.2), we directly utilized the SPE decrementer register in order to reduce the amount of instrumentation overhead. The value in this register is decremented at a rate of 79.8MHz, providing a precision of approximately 12ns. The transactional runtime system is configured with an ORT of 2^{20} records. The read and write sets are statically configured to 2^{13} and 2^{10} entries, respectively, in order to avoid overflows.

The STAMP [**?**] benchmark suite is used for evaluation purposes. Results are reported for 6 out of the 8 provided applications: (1) Genome, a DNA sequencing algorithm; (2) Intruder, a network intrusion detection application; (3) Kmeans, a clustering algorithm; (4) Labyrinth, an implementation of the Lee [31] algorithm for maze routing; (5) SSCA2, a graph construction application using adjacency arrays; and (6) Vacation, a simulation of a travel reservation system. We did not use the other two applications, Bayes and Yada, since Bayes exhibited nondeterministic behavior (confidence interval too large) and Yada did not produce correct results on our system. Others in the literature have faced the same problem [32, 33].

The configurations used for each application are given by Table 1. Note that some of them differ from the recommendations specified in the STAMP paper, mostly because of SPU restrictions such as memory size (recall that an SPE has only 256KB of local storage). For instance, choosing a larger maze

Application	Arguments
Genome	-g1024 -s64 -n131072
Intruder	-a20 -l896 -n1536 -s1
Kmeans	-m80 -n80 -t0.05 -i random-n16384-d24-c16
Labyrinth	-i random-x86-y86-z3-512
SSCA2	-s14 -i1.0 -u1.0 -l9 -p9
Vacation	-n4 -q60 -u90 -r524288 -t32768

Table 1: Application configurations.

size for Labyrinth would cause the program to crash when dynamic memory was requested. The arguments were selected so that the running time of the applications was reasonable and appropriate for the experiments. A short execution time (less than 1 second) would produce a large confidence interval and unreliable results. We characterize the chosen 6 STAMP applications with the given arguments in Section 5.2. Before that, however, we conduct a sensitivity analysis of the main transactional SMC parameters in the next Section.

5.1. SMC Analysis

Recall from Section 4.2 that our transactional runtime operates at the cache line granularity. Therefore, when a word is referenced in the code, the whole cache line is actually versioned. On the one hand, a larger cache line favors spatial locality and may reduce the cache miss rate. On the other hand, the probability of false conflicts due to multiple SPEs sharing the same cache line tends to increase.

The number of cache lines also favors locality and may help to improve system performance but, since the cache used by our system is implemented in software, there is an extra cost in flushing its contents (the cost is proportional to the number of cache lines). This is important to notice since a flush is performed every time a transaction is started. Therefore, there is also a tradeoff in selecting the quantity of lines for the SMC: a bigger value increases spatial and temporal locality, but an additional price is paid for flush operations.

In order to investigate the issues raised in the previous discussion, we conducted an analysis to decide the best cache configuration for each application. We are primarily interested in selecting the best cache line size and the best number of lines. Due to implementation issues, the SMC is always configured as a 4-way set associative cache.

Figure 5 shows the results for the 6 studied STAMP applications. The number of cache sets is varied from 2^3 to 2^7 (shown along the X axis) and, for each set size, the cache line size ranges from 2^4 to 2^7, covering a substantial fraction of the design space. The Y axis displays the execution time (in seconds) for each cache line size configuration. The results are for 6 SPEs.

Firstly, let us discuss what can be observed as the number of sets is increased. As we mentioned earlier, the cost of the flush operation increases linearly with the number of cache lines. If the performance gain resulted from a reduced miss rate cannot compensate the extra cost induced by the flushes, then we should see worse execution times. In fact, this very behavior is specially observed in applications Genome, Intruder and Kmeans. As can be confirmed by Table 2, the cache miss rate (CMR) decreases very slowly

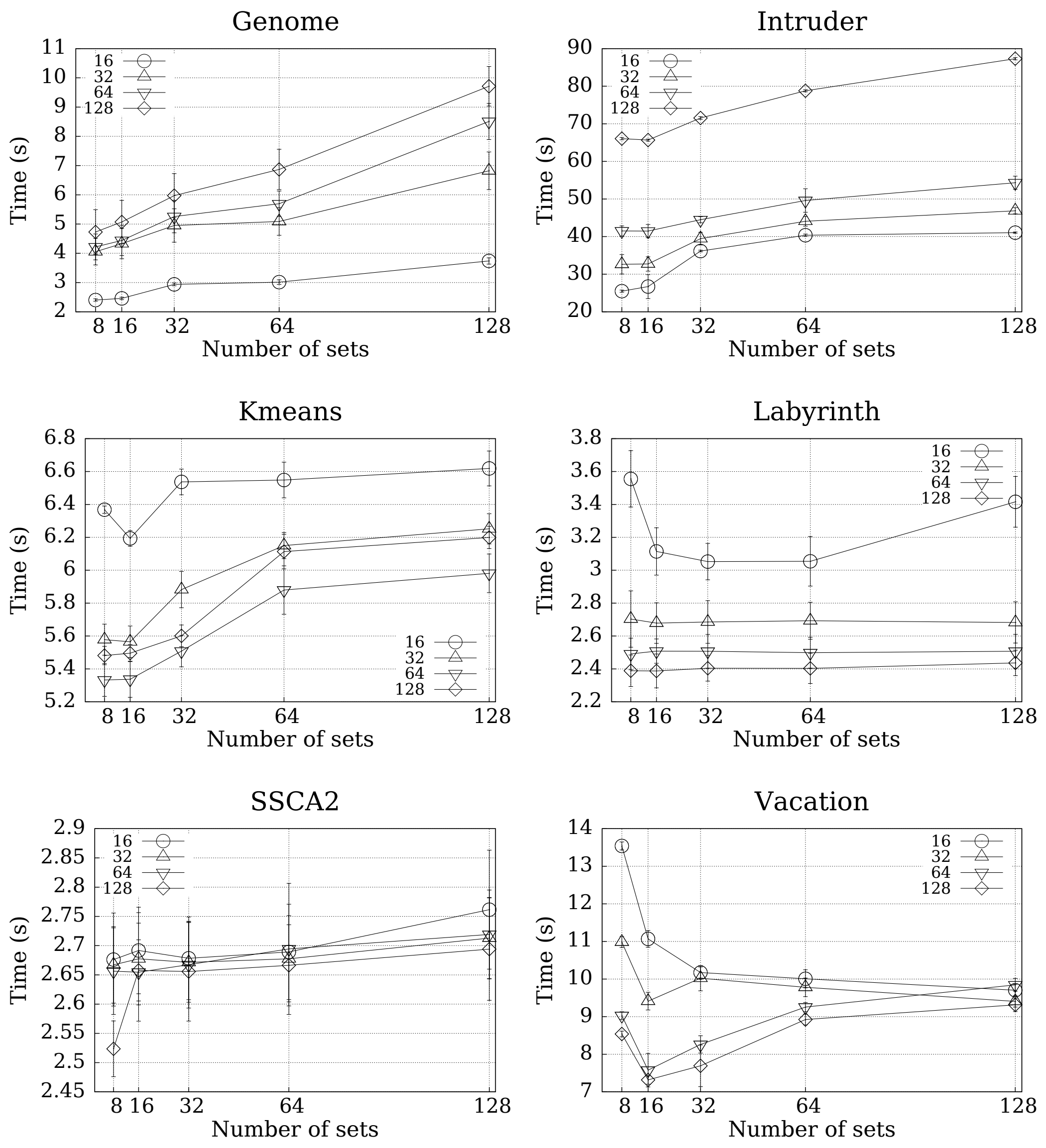

Figure 5: Runtime performance for different cache configurations.

as the set size is increased for these applications. Therefore, the cost added by the flushes is dominant.

On the other hand, application Vacation seems to perform better when the set size is increased from 8 to 16. Performance tends to decrease after that point, however. Table 2 shows that, for this application, the CMR is significantly reduced when 16 sets are used instead of 8, explaining the performance boost. Consider, for instance, cache lines of 128 bytes: the miss rate is reduced from 20% to 11% when the set size is increased from 8 to 16. Larger set sizes (32, 64, and 128) do not provide further improvements, as the flush cost starts to materialize. For other applications, such as Labyrinth and SSCA2, the set size does not seem to make much difference. We observe an improvement for Labyrinth with cache lines of 16 bytes when the set size is increased to 16 and 32. However, the performs tends to degrade after 64 sets.

We now turn our attention to the impact of different cache line sizes on the performance. Ideally, we would expect bigger line sizes to yield better performance. However, as we already pointed out, false sharing can somewhat mitigate the advantages. To shed some light into this issue we also present in Table 2 the number of retries per transaction (RPT). This value give us a hint of how many transactions are aborted per committed transaction. It is evident from Figure 5 that the performance for applications Genome and Intruder is severely hurt with the increased line size. Looking at Table 2 it is apparent that the number of aborting transactions increases with larger line sizes. For instance, increasing the line size from 16 to 32 bytes increases the RPT from almost 0 to 0.77 for the Genome application

Cache		Genome		Intruder		Kmeans		Labyrinth		SSCA2		Vacation	
		CMR	RPT	CMR	RPT	CMR	RPT	CMR	RPT	CMR	RPT	CMR	RPT
2^3	2^4	7.12	0.00	54.59	0.30	18.44	0.02	4.48	0.81	29.31	0.01	65.71	5.22
	2^5	4.83	0.77	50.14	0.42	7.76	0.52	1.87	0.74	21.44	0.01	36.13	3.99
	2^6	3.74	0.83	42.19	0.56	3.71	0.26	0.77	0.75	16.47	0.01	22.95	3.80
	2^7	3.28	1.00	36.41	0.95	1.98	0.57	0.38	0.69	13.30	0.01	20.33	3.63
2^4	2^4	6.44	0.00	50.81	0.31	16.41	0.02	3.70	0.75	28.51	0.00	26.22	4.63
	2^5	4.45	0.83	47.96	0.40	7.33	0.42	1.51	0.78	21.17	0.01	17.00	4.04
	2^6	3.50	0.88	40.44	0.55	3.57	0.20	0.74	0.72	16.29	0.01	12.30	2.76
	2^7	3.00	1.11	34.88	0.87	1.89	0.46	0.38	0.70	13.18	0.01	11.34	2.57
2^5	2^4	6.22	0.00	50.03	0.74	15.61	0.09	3.01	0.86	28.30	0.00	17.14	4.48
	2^5	4.37	0.87	46.91	0.64	7.16	0.21	1.49	0.82	21.08	0.01	13.56	5.06
	2^6	3.60	1.16	39.10	0.69	3.52	0.19	0.74	0.72	16.23	0.01	10.38	3.59
	2^7	3.27	1.49	34.21	0.88	1.85	0.40	0.37	0.68	13.14	0.01	9.43	3.27
2^6	2^4	6.14	0.00	48.34	0.53	15.23	0.05	2.98	0.86	28.20	0.00	15.47	4.21
	2^5	4.36	0.86	46.58	0.52	7.08	0.12	1.48	0.78	21.04	0.01	12.36	4.45
	2^6	3.63	1.37	38.94	0.57	3.47	0.18	0.74	0.71	16.20	0.01	10.26	4.65
	2^7	3.08	1.77	34.09	0.85	1.85	0.39	0.37	0.67	13.12	0.01	9.28	4.46
2^7	2^4	6.11	0.00	47.57	0.35	15.05	0.03	2.98	0.80	28.16	0.00	14.69	3.64
	2^5	4.17	0.85	45.95	0.38	7.00	0.10	1.49	0.75	20.99	0.00	11.81	3.67
	2^6	3.69	1.61	38.98	0.51	3.47	0.13	0.74	0.71	16.17	0.01	10.25	5.14
	2^7	3.14	2.01	34.07	0.87	1.84	0.30	0.37	0.70	13.12	0.01	9.20	3.99

Table 2: Average Cache Miss Rate (CMR) and Retries Per Transaction (RPT) for each cache configuration studied. Different set sizes are separated by a horizontal line for better visualization. The CMR is reported in percentage and is the combined sum of the read and write miss rates.

(set size 8). This seems to indicate that a false sharing scenario is inducing spurious aborts.

Kmeans is an interesting case. The best line size for this application is 64 bytes. Smaller sizes (16 and 32), as well as a larger one (128), tend to degrade performance. Looking at Table 2 we can indeed notice that the RPT value for the configuration of 64 bytes is smaller than the respective ones for configurations 32 and 128, specially for set sizes 8 and 16. More interestingly, the RPT is smaller for lines of 16 bytes, despite the fact that the performance of this particular configuration is the worst. As we will show in the next section, the total time Kmeans spends executing transactions is only about 35% of the total time. Therefore, we believe the impact of the RPT is not the dominant factor. A reduced CMR is probably giving more benefits than a lower RPT, although it is not clear from the table what is the best ratio. For the remaining applications, a larger line size yields better results.

The analysis conducted in this section is important because we can eliminate SMC configurations that could mask our performance results presented in Section 5.3. From this point on, we report results with the best SMC configurations: 8 sets of 128 bytes for SSCA; 16 sets of 16 bytes for Genome and Intruder; 16 sets of 64 bytes for Kmeans; 16 sets of 128 bytes for Vacation; and 32 sets of 128 bytes for Labyrinth.

5.2. Benchmark Characterization

As discussed earlier, it was not possible to use the recommended configurations for all the selected STAMP applications. In order to enhance the confidence of our results, we also instrumented the code to measure some of the most important transactional characteristics for each application.

Table 3 presents, for each application, the averages for transaction length (in μs), total time spent in transactions, read and write set sizes (in cache lines) and the number of retries per transaction (RPT). We can notice that the applications cover different execution scenarios. For instance, the average transaction length varies from 870ns (SSCA2) to 7.7ms (Labyrinth). Total time fraction spent inside transactions is also very diversified, ranging from 35% (Kmeans) to 98% (Labyrinth).

The size of the read and write sets tends to increase with the transaction length. For instance, the application with longer transactions (Labyrinth) also has a bigger read set (450 entries). As expected, the number of transactional reads is consistently larger than the number of writes. The quantity of retries per transaction characterizes different contention levels. Vacation, in particular, shows a high amount of aborts per committed transaction, whereas for Genome and SSCA2 this value is very low. Observe that, although the RPT for Genome shows 0.0, this does not mean that no transaction is ever aborted in this application. What it states is that the number of aborts per committed transaction is extremely low.

When we compare the results from our characterization with the one provided in the STAMP paper [**?**], we can see that they are very consistent. We only observed a substantial discrepancy in the total time in transactions for Vacation. We believe this is due to the different configuration we are using for this application. The difference could also be explained by the fact that the results reported in the STAMP paper were collected in a simulated environment.

Application	Tx length (μs)	Tx time	Read set	Write set	RPT
Genome	106.04	87.3%	34.8	2.4	0.00
Intruder	45.46	97.8%	74.0	1.15	0.31
Kmeans	3.54	35.5%	4.3	1.37	0.20
Labyrinth	7680.77	98%	450.5	13.8	0.68
SSCA2	0.87	85.3%	3.0	2.0	0.01
Vacation	92.26	59%	39.8	0.9	2.57

Table 3: Benchmark characterization.

5.3. Performance

To assess the performance of our prototype runtime system we compare the transactional version of each application to a lock-based version, in which a global lock is acquired at the beginning of a transaction and released at the commit point. In other words, we simply replace `begin_atomic` and `end_atomic` with operations to acquire and release a global lock. The SMC configuration is kept unchanged for the lock-based version of the applications and also uses the best parameters as determined in Section 5.1.

Results are shown normalized to the sequential version of the same application executed on a single SPE. We do not compare the performance with the sequential PPE version since, as discussed in Section 4.7, our runtime system is not yet optimized for the SPE. Also, we did not attempt to optimize any of the STAMP applications for the SPE. Therefore, comparing the SPE execution time with the respective PPE time would be inappropriate. On the other hand, since our results are normalized with regard to the sequential version executed on the SPE, any optimization applied to this version will automatically affect the performance of the transactional version.

Figure 6 shows performance results as the number of SPEs is increased from 1 to 6 (recall that only 6 SPEs are available in our PS3 machine) for the six selected STAMP applications. First of all, notice that practically all transactional applications have a high overhead when only a single SPE is employed, with the exception of Kmeans, which spends only 35% of the total time in transactions.

With only one SPE, the overhead added by the read and write barriers, along with the commit operation, is too high. As the number of SPEs is increased, we hope that the amount of useful work performed by the system as a whole will also increase, and compensate the overhead imposed by the transactional primitives. Therefore, we expect to see some improvement as more SPEs are executed in parallel.

Indeed, our system shows good scalability for applications Genome, Intruder, and Labyrinth. The transactional versions of Genome and Labyrinth outperform the lock-based counterparts as soon as there are 2 concurrent SPEs running. For Labyrinth, 2 SPEs are enough for the transactional version to also surpass the sequential version, and a speedup of 1.5x is obtained with 6 SPEs. Genome and Intruder also presented better performance than the sequential version starting from 4 and 6 SPEs, respectively. Also, notice that the lock-based version does not scale for any of these applications.

The lock-based version of Kmeans already displays an almost perfect scalability. Therefore, this application is probably not the best candidate for transactions. However, the transactional version also scales, albeit at a slower rate due to the extra overhead. As for the two remaining applications, they do not seem to benefit from our runtime system. In order to

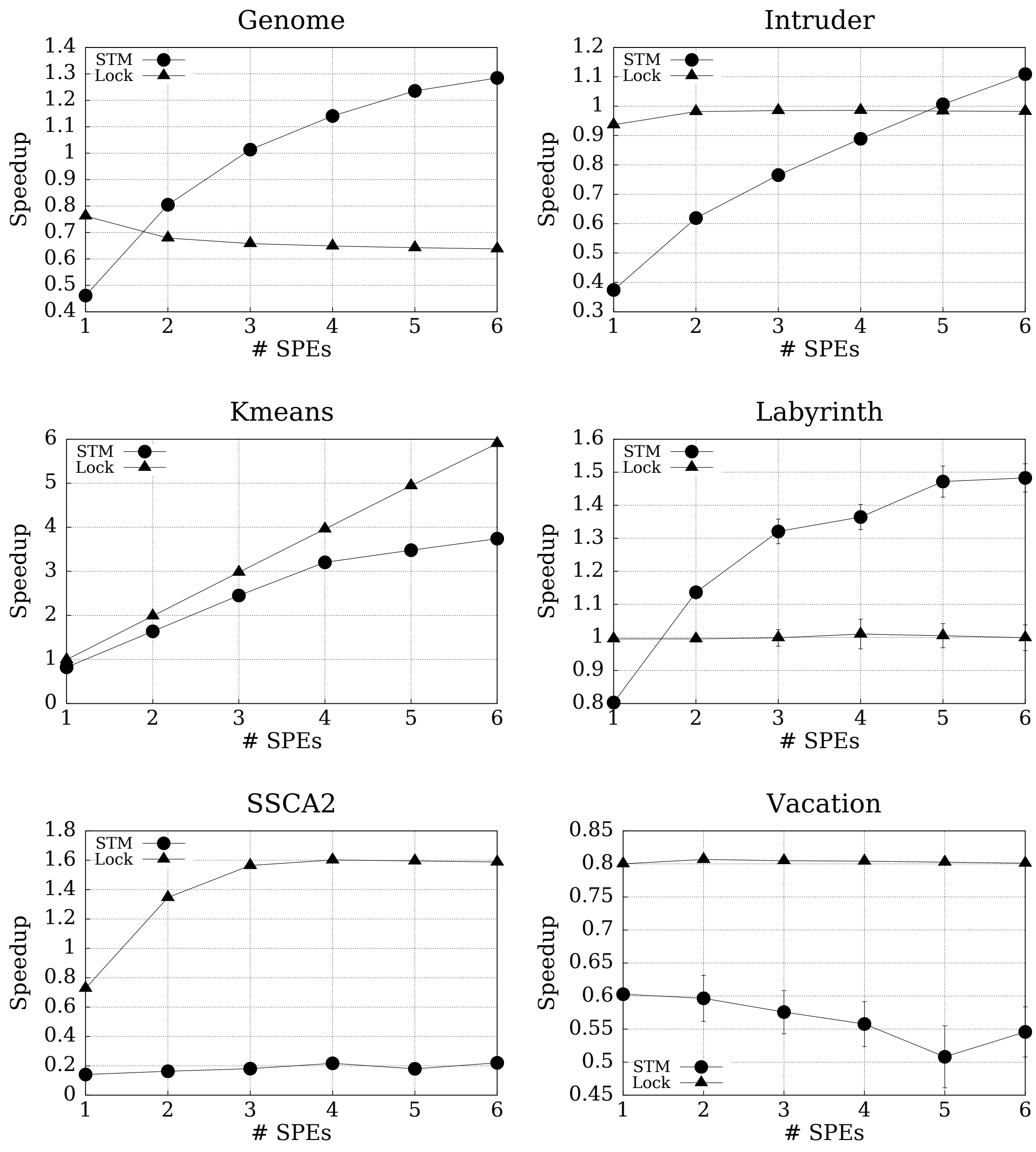

Figure 6: Performance results (normalized with regard to the sequential version executed with a single SPE).

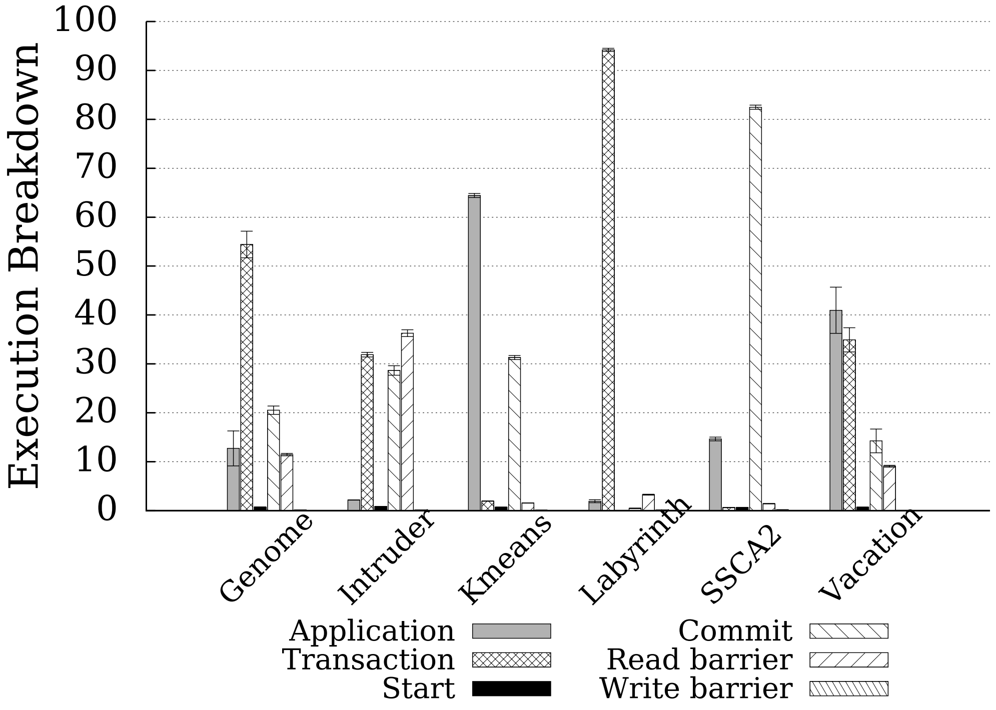

Figure 7: Time breakdown.

further investigate the reasons for the performance loss we re-executed the experiments with instrumentation added to measure the overhead of each transactional primitive. Figure 7 depicts the fraction of time for each of the main primitives, along with the time spent outside transactions (Application) and the time spent by transactions without the overhead of the primitives (Transaction). The bar named “Commit” represents the time spent on both successful and failed commits. A configuration with 6 SPEs is used to collect the data for this figure.

It can be seen from Figure 7 that SSCA2 spends the majority of its time doing commits. In fact, this application has extremely small transactions

(870ns) that execute only a small number of reads and writes per transaction. Therefore, the cost to execute the commit operation prevails over the actual time spent executing the transaction itself. The lock-based version, on the contrary, seems to provide a performance gain over the sequential version. The SSCA2 application characterizes the worst case scenario for our runtime system. Applications with very small critical regions will probably get more benefit from locks than software transactions.

Vacation spends a large fraction of time outside transactions and, most importantly, exhibits a high quantity of retries per transaction (the highest of the studied applications). Consequently, most of the time we see in the "Transaction" bar is actually due to the re-execution of aborted transactions. Since the amount of useful work is low, the application does not scale with the use of transactions, nor does it with locks. Vacation seems to indicate that applications with a high level of contention are not the best candidates for our runtime. This is understandable, since our runtime adopts an optimistic approach to concurrency which works better when conflicts are not common.

Figure 7 also elucidates why applications Genome, Intruder and Labyrinth display the best results with the transactional runtime. Notice that the actual transactional time for these applications, not counting the overhead, is higher than the respective time for the other applications (55%, 32%, and 94%, respectively) [1]. Particularly, Labyrinth spends most of its time with only transactions and achieves a relatively good speedup (1.5x). Moreover, Figure 7 indicates that the read barrier and the commit operation have the

[1] Vacation actually spends most of its time re-executing aborted transactions and therefore is not considered.

highest overhead, and should be the primary target for optimizations.

5.4. Discussion

The overhead added by software transactional memory runtimes is well known for homogeneous architectures and has attracted a lot of criticism, to the point of STM being regarded as a *research toy* [34]. Recently, a thorough investigation of STM performance showed that STM can outperform sequential code with four threads on 13 of the 17 studied applications on an x86 system [35]. One key point when evaluating transactional behavior is how well the system scales. For instance, compiler instrumentation may hurt performance, but will not affect the scalability [35].

In this section we discuss how our results compare with the ones generally obtained for homogeneous architectures and argue that the proposed runtime for a heterogeneous architecture displays similar characteristics, therefore providing evidence for its effectiveness. Figure 8 depicts the speedup over sequential execution of the six studied applications using the TL2 implementation provided with the STAMP benchmark. The results were collected with a dual Xeon E5405 machine (8 cores available), 4GB of RAM, running at 2GHz in a typical 32-bit Linux operating system. We refer to this configuration as the *x86 machine* in the following discussion. We also use *PS3 machine* for the heterogeneous system. The number of cores employed in the x86 experiments varies from 1 to 8.

The first characteristic that both results from Figure 6 (PS3 machine) and Figure 8 (x86 machine) share is the high single core overhead. For instance, the performance loss is about 70% for the homogeneous Vacation and 80% for the heterogeneous SSCA2. With only two exceptions (homogeneous

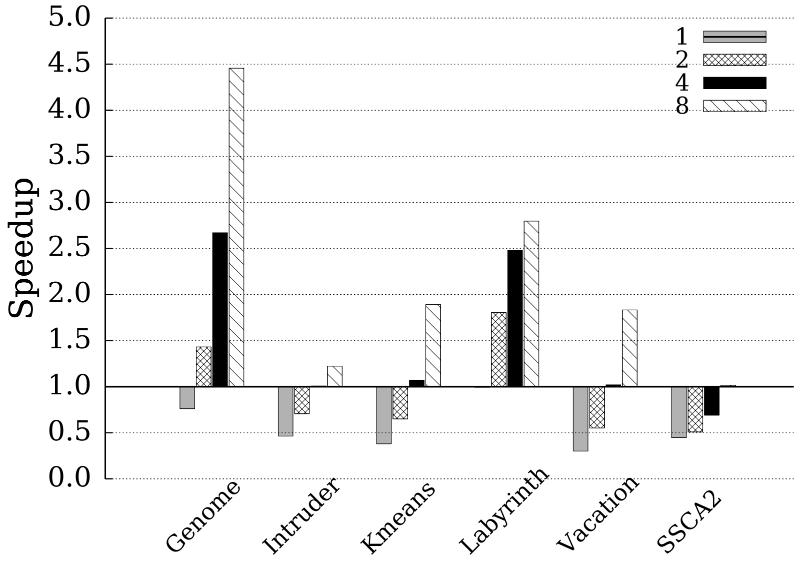

Figure 8: Speedup of the studied applications on a dual quad-core machine.

Labyrinth and heterogeneous Kmeans), all applications show significant performance losses in the single core case.

Figure 8 also reveals that the transactional code starts to outperform the sequential code when 8 cores are used. If 4 cores are considered instead, only Genome and Labyrinth produce a substantial speedup. Therefore, it can be assumed that the minimum number of cores from which the x86 machine shows its real benefits is 8. If we examine the PS3 machine results (Figure 6) we can notice a similar trend: with 6 cores, 4 out of the 6 applications exhibit better performance with the transactional runtime. The only two exceptions are SSCA2 and Vacation since, as explained in the previous section, they either have extremely small transactions (SSCA2) or a high retries per transaction ratio (Vacation). This behavior is actually the main performance difference between the machines: the cost to initialize and commit/abort a transaction is higher on the PS3 machine due to its segregated memory architecture and communication penalties.

Looking at the actual speedup numbers we can see that they are more favorable to the x86 machine. For instance, the speedup of Labyrinth with 4 cores is 2.5x on the x86 and 1.5x on the PS3 machine with 6 cores. Besides the distinct architecture, there are at least two reasons for this. First, the x86 results (Figure 8) use a manually instrumented version of the STAMP benchmark, whereas our approach uses a compiler to instrument the code. Second, as briefly discussed in Section 4.7, our prototype implementation is not tuned for performance. Therefore, we envision a number of different paths to improve its efficiency: (i) compiler optimizations to reduce the number of compiler-induced overhead as well as the memory ordering overhead, such as suggested in [12, 36]; (ii) SPE-specific optimizations: average speedups of 9.9x are reported in the literature with simdization [30].

Of all the aspects considered in the comparison with the x86 machine, we believe that scalability is the most important one. Figure 8 shows that all x86 applications scale well, even the one that did not achieve concrete speedup (SSCA2). Likewise, our results from Figure 6 show that, with the exception of SSCA2 and Vacation, the PS3 applications exhibit good scalability. Given the fact that our system uses a compiler-assisted approach and that compiler instrumentation may degrade performance (but not scalability) [35], we believe that the obtained results are promising.

6. Conclusions

Designing heterogeneous architectures is one promising approach in the multicore era, but the lack of appropriate tools and programming models have made it difficult to develop new multithreaded applications for such hardware.

In order to facilitate the development of concurrent software, we investigated in this article the transactional model and presented the design and implementation of a transactional runtime system for the Cell/BE architecture. Our system provides compiler support so that read and write barriers are automatically inserted into the code, reducing coding time and mistakes.

We evaluated a prototype implementation of the proposed system using the STAMP benchmark suite and obtained promising results. More precisely, our system exhibited good scalability for applications with moderate to low contention levels, and whose transactions are not too small. Moreover, we believe that a small performance loss is justified given the ease of programming provided by the transactional model.

7. Acknowledgements

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

References

[1] K. Olukotun, L. Hammond, The future of microprocessors, Queue 3 (7) (2005) 26–34.

[2] R. Kumar, D. M. Tullsen, N. Jouppi, P. Ranganathan, Heterogeneous chip multiprocessors, IEEE Computer 38 (11) (2005) 32–38.

[3] C. R. Johns, D. A. Brokenshire, Introduction to the Cell broadband engine architecture, IBM Journal of Research and Development 51 (5) (2007) 503–519.

[4] T. Chen, R. Raghavan, J. N. Dale, E. Iwata, Cell broadband engine architecture and its first implementation: A performance view, IBM Journal of Research and Development 51 (5) (2007) 559–572.

[5] T. Harris, J. Larus, R. Rajwar, Transactional Memory, 2nd Edition, Morgan & Claypool Publishers, 2010.

[6] J. Larus, C. Kozyrakis, Transactional memory, Communications of the ACM 51 (7) (2008) 80–88.

[7] D. Dice, O. Shalev, N. Shavit, Transactional Locking II, in: 20th International Symposium on Distributed Computing, 2006, pp. 194–208.

[8] C. C. Minh, J. Chung, C. Kozyrakis, K. Olukotun, STAMP: Stanford Transactional Applications for Multi-Processing, in: Proceedings of the IEEE International Symposium on Workload Characterization, 2008, pp. 35–46.

[9] M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, Software transactional memory for dynamic-sized data structures, in: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing, 2003, pp. 92–101.

[10] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer, M. L. Scott, Lowering the overhead of nonblocking software

transactional memory, in: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, 2006.

[11] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, B. Hertzberg, McRT-STM: A high performance software transactional memory system for a multi-core runtime, in: Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming, 2006, pp. 187–197.

[12] T. Harris, M. Plesko, A. Shinnar, D. Tarditi, Optimizing memory transactions, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2006, pp. 14–25.

[13] R. Guerraoui, M. Kapalka, On the correctness of transactional memory, in: Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming, 2008, pp. 175–184.

[14] P. Felber, C. Fetzer, T. Riegel, Dynamic performance tuning of word-based software transactional memory, in: Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming, 2008, pp. 237–246.

[15] A. Dragojevic, R. Guerraoui, M. Kapalka, Stretching transactional memory, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, 2009, pp. 155–165.

[16] IBM, Software development kit for multicore acceleration version 3.1, programmer's guide (2008).

[17] J. Balart, M. Gonzalez, X. Martorell, E. Ayguade, Z. Sura, T. Chen, T. Zhang, K. O'Brien, K. O'Brien, A novel asynchronous software cache implementation for the Cell-BE processor, in: 20th International Workshop on Languages and Compilers for Parallel Computing, 2008, pp. 125–140.

[18] M. Gonzalez, N. Vujic, X. Martorell, E. Ayguade, A. E. Eichenberger, T. Chen, Z. Sura, T. Zhang, K. O'Brien, K. O'Brien, Hybrid access-specific software cache techniques for the Cell BE architecture, in: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, 2008, pp. 292–302.

[19] G. Senthil, S. Gudla, P. K. Baruah, Exploring software cache on the Cell BE processor, in: International Conference on High Performance Computing, 2008.

[20] S. Seo, J. Lee, Z. Sura, Design and implementation of software-managed caches for multicores with local memory, in: Proceedings of the 15th International Symposium on High-Performance Computer Architecture, 2009, pp. 55–66.

[21] T. Chen, T. Zhang, Z. Sura, M. G. Tallada, Prefetching irregular references for software cache on Cell, in: Proceedings of the International Symposium on Code Generation and Optimization, 2008, pp. 155–164.

[22] J. Lee, S. Seo, C. Kim, J. Kim, P. Chun, Z. Sura, J. Kim, S. Han, COMIC: A coherent shared memory interface for Cell BE, in: Proceed-

ings of the 17th International Conference on Parallel Architectures and Compilation Techniques, 2008, pp. 303–314.

[23] M. Olszewski, J. Cutler, J. G. Steffan, JudoSTM: A dynamic binary-rewriting approach to software transactional memory, in: Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques, 2007, pp. 365–375.

[24] M. F. Spear, M. M. Michael, C. von Praun, RingSTM: Scalable transactions with a single atomic instruction, in: Proceedings of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures, 2008, pp. 275–284.

[25] L. Dalessandro, M. F. Spear, M. L. Scott, NOrec: Streamlining STM by abolishing ownership records, in: Proceedings of the 15th Symposium on Principles and Practice of Parallel Programming, 2010, pp. 67–78.

[26] R. L. Bocchino, V. S. Adve, B. L. Chamberlain, Software transactional memory for large scale clusters, in: Proceedings of the 13th Symposium on Principles and Practice of Parallel Programming, 2008, pp. 247–258.

[27] C. Kotselidis, M. Ansari, K. Jarvis, M. Lujan, C. Kirkham, I. Watson, DiSTM: A software transactional memory framework for clusters, in: Proceedings of the 37th International Conference on Parallel Processing, 2008, pp. 51–58.

[28] J. Lee, S. Seo, J. Lee, A software-SVM-based transactional memory for multicore accelerator architectures with local memory, in: Proceedings

of the 19th International Conference on Parallel Architectures and Compilation Techniques, 2010, pp. 567–568.

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

[30] 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, M. Gschwind, Optimizing compiler for the CELL processor, in: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, 2005, pp. 161–172.

[31] C. Y. Lee, An algorithm for path connections and its applications, IRE Transactions on Electronic Computers EC-10 (3) (1961) 346–365.

[32] A. Roy, S. Hand, T. Harris, A runtime system for software lock elision, in: Proceedings of the 4th European Conference on Computer Systems, 2009, pp. 261–274.

[33] D. Christie, J.-W. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, E. Riviere, Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack, in: Proceedings of the 5th European Conference on Computer Systems, 2010, pp. 27–40.

[34] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, S. Chatterjee, Software transactional memory: Why is it only a research toy?, Communications of the ACM 51 (11) (2008) 40–46.

[35] A. Dragojevic, P. Felber, V. Gramoli, R. Guerraoui, Why STM can be more than a research toy, Communications of the ACM 54 (4) (2011) 70–77.

[36] M. F. Spear, M. M. Michael, M. L. Scott, P. Wu, Reducing memory ordering overheads in software transactional memory, in: Proceedings of the International Symposium on Code Generation and Optimization, 2009, pp. 13–24.