Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

tlCell: a Software Transactional Memory for the Cell Broadband Engine Architecture

André Filipe da Rocha Lopes — 26949

Lisboa
(2010)

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

tlCell: a Software Transactional Memory for the Cell Broadband Engine Architecture

André Filipe da Rocha Lopes — 26949

Orientador: Prof. Doutor João M. S. Lourenço

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para a obtenção do Grau de Mestre em Engenharia Informática.

Lisboa
(2010)

Dedicado à minha familia e amigos.

Acknowledgements

First of all I would like to thank Prof. Doutor João Lourenço for his guidance during the development of this Dissertation but specially for his pedagogic and human skills, which are remarkable. Without him it would had been a much more difficult journey. Also a very special thanks to all the Professors of Computer Science Department which during the development of this Dissertation had the time to help me whenever i needed, namely Prof. Doutor Hervé Paulino and Prof. Doutor Pedro Barahona for having time to discuss some technical issues of Cell Broadband Engine and also Prof. Doutor Paulo Lopes for restarting the blade server every time I managed to deadlock it.

Also a very special thanks to Prof. Doutor José Cardoso e Cunha to which I have the greatest respect and admiration and helped me follow this field of study when taking one of his subjects.

To all my colleagues, which stood by my side all the time and spent numerous hours in Computer Engineer Department with me, a very special thanks.

A very special thanks to my family, for guidance, support and patience.

This work was partially supported by Sun Microsystems and Sun Microsystems Portugal under the “Sun Worldwide Marketing Loaner Agreement #11497”.

Summary

The evolution of computers grew exponentially in the last decades. The performance has always been the main concern resulting in increasing clock frequency of processors, which is not feasible anymore due to power consumption of actual energy-starving processors. Cell Broadband Engine Architecture project started with the goal of delivering high performance with low power consumption. The result is a heterogeneous multiprocessor architecture with a unique memory design space towards high performance and reduced hardware complexity to reduce the cost of production. In such an architecture it is expected that concurrency and parallelism techniques improve performance substantially. However the high performance solutions presented for CBEA are very specific due to its novel architecture and memory distribution and it is still hard to develop tools that are able to provide to the programmer an abstraction layer that is able to exploit concurrency and manage consistency. Software Transactional Memory is a programming model that proposes this abstraction layer, and is gaining increased popularity and several prototypes have been developed with performance close to fine-grain specific implementations for the domain problem. The possibility of using STM to develop a tool capable of hiding all the memory management and consistency in CBEA is very appellative. In this document we specify a deffered-update STM framework for CBEA that takes advantage of the SPEs for computational power using a commit-time locking mechanism for commiting transactions. Also two different models are proposed, fully local and multi-buffered models in order to better study the implications of our design choices.

Keywords: Software Transactional Memory, Cell Broadband Engine Architecture, Consistent Transaction Layer.

Sumário

Os computadores evoluíram exponencialmente na ultima década. A performance tem sido o principal objectivo resultando no aumento do frequência dos processadores, situação que já não é fazível devido ao consumo de energia exagerado dos processadores actuais. A arquitectura Cell Broadband Engine começou com o objectivo de providenciar alta capacidade computacional com um baixo consumo energético. O resultado é uma arquitectura com multi-processadores heterogéneos e uma distribuição de memória única com vista a alto desempenho e redução da complexidade do hardware para reduzir o custo de produção. Espera-se que as técnicas de concorrência e paralelismo aumentem a performance desta arquitectura, no entanto as soluções de alto desempenho apresentadas são sempre muito especificas e devido à sua arquitectura e distribuição de memória inovadora é ainda difícil apresentar ferramentas passíveis de explorar concorrência e paralelismo como um camada de abstracção. Memória Transaccional por Software é um modelo de programação que propõe este nível de abstracção e tem vindo a ganhar popularidade existindo já variadas implementações com performance perto de soluções específicas de grão fino. A possibilidade de usar Memória Transaccional por Software nesta arquitectura inovadora, desenvolvendo uma ferramenta capaz de abstrair o programador da consistência e gestão de memória é apelativo. Neste documento especifica-se uma plataforma deffered-update de Memória Transactional por Software para a arquitectura Cell Broadband Engine que tira partido da capacidade computacional dos Synergistic Processing Elements (SPEs) usando locks em commit-time. São propostos dois modelos diferentes, *fully local* e *multi-buffered* de forma a poder estudar as implicações das escolhas feitas no desenho da plataforma.

Palavras-chave: Memória transacional por Software, Cell Broadband Engine Architecture, Consistent Transaction Layer.

Contents

List of Figures

Listings

1

Introduction

1.1 Motivation

As the need for high demanding processing in almost every aspect of our society grows, Cell Broadband Engine Architecture (CBEA) was designed to provide a very high performance with low power consumption. Cell Broadband Engine (CBE) appears as a very peculiar type of architecture, leading to the need to develop brand new frameworks, compilers and runtime management tools, therefore implying the need to redesign the application code to achieve the expected performance from such an architecture.

Being an alternative to conventional systems, it is hard to expect from the common programmer the knowledge to code in CBE taking out of the architecture the real potential of it. This represents a massive effort to provide tools to the programmers so that they can abstract from the architecture design. Due to its heterogeneous multiprocessor architecture, concurrency and parallelism techniques are expected to greatly improve the speedup of the application code, but they also represent increased difficulty in its implementation, consistency and debugging.

Software Transactional Memory (STM), due to its properties such as being time-line execution independent and providing the concepts of atomicity and isolation, presents an approach to this high-level abstraction tools for the programmer on CBEA. With transactions, the programmer is able to define computations inside the scope of a transaction and benefit from the Transactional Manager to avoid inconsistency in the execution of code and relieves the programmer from the need of explicitly controlling the contention on the objects being accessed in the critical sections.

With these premises STM can offer great advantages in such novel architecture with still complex synchronization mechanisms, hoping to provide an abstraction layer to the programmer.

1.2 Problem specification

Most STM engines implement transactional memory for threads executing in the address space of a single computer. Exceptions, such as XSTM [Noe] and the distributed multiversioning STM engine presented by [MMA06], refer to a distributed shared memory system. A STM engine for CBEA does not suit the normal parameters of a simple shared memory system since the sub-processors, the Synergistic Processing Element (see section 2.4.1 for more details), cannot address directly the memory. All data transfers between the sub-processors and shared memory have to be made through Direct Memory Access (DMA) transfers (see section 2.4.2 for more details). Also, the SPE's are Single Instruction Multiple Data (SIMD) processors, with distinguish Instruction Set Architecture (ISA) from the main processor, which arises the need to have special concerns on the specification of a STM framework for the CBE to allow users to take advantage of the massive computational power of these processors.

Also the versatility of the CBEA allows to use it for several different programming paradigms which makes the task of developing and evaluating an STM framework for this architecture complex but also challenging. Providing several distinct synchronization methods, an heterogeneous multi-processor architecture, a novel memory distribution, a high capacity Bus (Element Interconnector Bus), the distinct compilers, frameworks and open-source libraries available the possibilities are almost infinite.

1.3 Thesis statement and contributions

With this study we intend to show that Transactional Memory paradigm fits CBEA.

The expected contribution is the specification and development of a STM engine for CBE under Linux operating system that allows integration with different programming models. We use Consistent Transaction Layer [Cun07] (also known as CTL) as a basis, due to its feature richness and the good performance achieved on x86 architecture. The integration of CTL on CBEA has several modifications, respecting to the different architecture design, supporting the heterogeneous multiprocessor architecture, its different instruction set architecture (ISA) and taking advantage of the unique memory layout and high memory bandwidth available.

The prototype implements a STM engine for CBEA, and it is benchmarked and tested to certify its correctness execution and performance achieved. We expect to contribute to the scientific community with the development of this prototype, providing detailed documentation and presenting the results achieved.

1.4 Document layout

This document is divided in 6 chapters.

- Chapter one introduces the problem, discusses the motivation and presents a brief solution layout.
- In chapter two we present related work. We describe the actual state with STM programming model, with an introduction to Transactional Memory properties and features, followed by a description of CTL. Also in chapter two, the Cell Broadband Engine and its functionalities are discussed, as well as CBEA and x86 ISA relevant differences.
- Chapter three presents the architecture solution for the specific problem, discussing the pros/cons of design choices on the provided solution.
- Chapter four discusses the implementation issues of the framework.
- In chapter five the validation and benchmarking of the prototype are presented. Also the results are discussed.
- Chapter six draws last conclusions on the development of this dissertation and discusses open issues that may be added in a future work.

2

Related Work

2.1 Introduction

In this chapter it is introduced the state of the technologies used in the study of this Dissertation. The first section introduces Software Transactional Memory (STM) and its properties. Afterwards Consistent Transaction Layer (CTL), a STM framework developed for x86 processor, is introduced and its properties explained since we use this framework as a basis for our prototype.

The following section explains the Cell Broadband Engine Architecture (CBEA), the design, properties and communication mechanisms that the architecture provides. Afterwards a comparison between x86 processor and Cell Broadband Engine Instruction Set Architecture (ISA) is made in order to understand the distinctions between them. The following section introduces Software Managed Caches (SMC) since it can greatly improve the performance of our solution in the specific architecture (CBEA). The last section introduces STM benchmarks, the specific problems in benchmarking STM application and some of the solutions and models provided in the scientific community in order to effectively and accurately benchmark STM implementations.

2.2 Software Transactional Memory

Transactions are well known in the world of Databases, they are in great part responsible for the success of Database programming model. Transactions in databases provide a way to execute queries of code concurrently maintaining consistency of persistent memory. Applying this transactional model to other programming models is becoming of great importance, specially with the current growing of multi-core processors. Single-chip processors have reached a limit in performance, it is not feasible anymore to increase the clock frequency, if we consider performance per energy consumed, and this evolution leads us to the present state, where applications and systems are not prepared for this multi-core or multi-processor architectures. Providing tools that help to exploit concurrency and parallelism has revealed to be a very demanding task [LR06].

Writing concurrent programs is not trivial [Han77], and is mostly based on explicit locking of critical sections where concurrent data accesses may occur and it is always implementation dependent. Transactional memory was proposed as a new abstraction for programming [Lom77], providing a high level abstraction for concurrency control in a multi-threaded or multi-processor environment. Being able to consider several pieces of code as transactions and executing them concurrently as if they were being executed sequentially brings notorious benefits.

Transactional Memory (TM) shifts the burden of synchronizing and coordinating parallel computation from the programmer to the TM framework, and the performance results are frequently equivalent or closed to fine grain lock based implementations. Although transactions are not the only way to control parallel computation (we still have locks, semaphores, mutexes and monitors) the higher-level abstraction sure makes transactional memory very appellative. This idea came from Lomet [Lom77] in 1977, who did not present any practical implementation until Herlihy and Moss proposed a hardware supported transactional memory [HEM93] in 1993. The term STM was first introduced by Shavit, and Touitou in 1995 [ST95] and described the first software implementation of a transactional memory.

In the sub-sequent sections it will be introduced the transactional model properties and design approaches for software transactional model, specifying the advantages and disadvantages of the different possibilities.

2.2.1 Transactional Model Properties

ACID properties

Transactions use the ACID(Atomicity, Consistency, Isolation and Durability) properties as fundamentals.

- **Atomicity** — This property states that either all the transaction instructions return successful, and therefore the transaction commits successfully, or if one of the instructions fail, the whole transaction aborts and none will be executed.

- **Consistency** — Consistency is directed related to data, specifically to data handled by a transaction. It is important that a transaction leaves a consistent data when it ends (either an abort or a commit). Every transaction expects reading a consistent state.

- **Isolation** — This specific property is very important for parallel environments. Each transaction has to run correctly even with other transaction executing concurrently, not affecting its data. Frequently it is assumed that the result of two concurrent transactions must be the same as if they were executed serially.

- **Durability** — After a transaction commits, the changes have to be persistent, even in case of system crash.

Usually transactions are closely related with databases, which are context specific, they interact with I/O devices and deal with persistent writes, where the property *Durability* is very important to assure persistent memory correctness. In STM, *Durability* is not taken into account, since we are dealing with memory writes/reads and data in memory is usually transient.

Transactions States

In the programmer perspective, there are three possible outcomes for a transaction. A transaction might commit; abort; or run in a undefined state [SKS05]. A transaction commits when all the instructions of the transaction have been successful. It aborts when any of the instructions fails. It might be undefined if it neither aborts or commits which results in a undefined or unhandled error on the execution of the transaction.

From the point of view of the system the transaction might be *active*, *partially committed*, *committed*, *failed* or *aborted*. An active transaction refers to a running transaction. In case of instructions completion (those inside the scope of a transaction) the transaction will try to commit, it is considered *partially committed*. If all changes were successful then the transaction is considered *committed* In case of any failure, the transaction must rollback, therefore encountering itself in a failed state, which is succeeded by the Aborted state, referring to the completion of the rollback leaving the data consistent.

Basic Constructs

An atomic block represents the blocks of code that will be executed as transactions. An example of this can be seen below.

Listing 2.1: Atomic Blocks

```
int calculate_square_area(int x, int y){
 atomic{
  int z = multiply(x,y);
 }
}
```

All the operations inside an atomic block have the *Isolation* and *Atomicity* properties assured and any function called inside an atomic block inherit the same properties.

Nested transactions

Nested transactions are transactions executed inside transactions. The original transaction is called the outer transaction, and the sub-transactions are the inner transactions. Several different approaches are made towards nested transactions behavior. This differences are how the transaction should handle the inner transactions respecting to data, and when aborting if the outer transaction should abort or not. Nested transactions were introduced by Moss [Mos81]. There are three types of Nested Transactions:

- *Flattened Nesting* — Flattened transactions behave, as the name says, as if the inner transaction expanded into the outer transaction. Aborting the inner transaction makes the outer transaction to abort as well and committing the inner transaction has no effect until the outer transaction commits. Data from a inner transaction is visible to the outer transaction, but not to the other transactions. These are easy to implement but a very simplistic way to approach nesting, but they can be used in a high rate commit scenario, where performance is critical.

- *Open Nesting* — Open nesting most specific attribute is that the result of the changes made by a inner transaction is made visible, not only to the outer transaction, but also for the rest of the system. Notice that even if the outer transaction aborts, the results altered by the previously committed inner transaction are still visible.

- *Closed Nesting* — Closed transactions aborts without aborting the outer transaction. When it commits passes control to the outer transactions, and changes made by the inner transaction become visible to parent transaction, but not to the rest of the transactions.

Distributed transactions

Distributed transactions happen often in the context of large databases, where data might be replicated through distributed data centers. To maintain certain properties, like atomicity, each agent must coordinate between them. A common approach for this is the Two-Phase Commit protocol. Each transaction acts as an agent, and whenever there is the need to commit a transaction the first phase of the protocol begins. The transaction responsible for the commit, lets say the manager, asks each agent the request to commit. After each agent "approve" the request, which implies that they execute the transaction to the point where they are able to commit, the responsible for the protocol initiation then starts the second phase, which is the effective commit of the changes, and afterwards the release of the locks. If any of the agents does not succeed in any of the phases, it informs the manager, which will message all the agents that the protocol has failed and therefore no changes will be made.

This protocol is widely known and used, but the fact that holds the locks while the protocol is in execution is a drawback since it minimizes concurrency and strongly affects performance.

2.2.2 Design approaches

Concurrency Control and Conflict Detection

Concurrency Control is necessary to synchronize concurrent accesses to an object/data.

When dealing with concurrent transactions sooner or later a conflict will happen. By conflict we mean that at least two transactions are accessing the same data, and one of them is trying to modify the data.

There are three possible states. A conflict occurs, a conflict is detected and a conflict is resolved. An occurrence of a conflict doesn't mean that it is immediately detected. We consider a conflict resolved when a TM rules/protocol is applied to resolve the conflict.

Resolving a conflict can be made in several ways. It varies on concurrency control mechanism of the TM namely if it is a pessimistic or optimistic concurrency control mechanism. In a pessimistic concurrency control approach, all the three stages happen sequentially. As soon as a conflict happens, it is detected and resolved. In a optimistic concurrency approach the same is not true. Conflict detection and resolution can be delayed.

Conflict Detection can be made at one of these times [LR06].

- Detected on open happens when we try to access the object or data at the first reference to the object.
- Detected on validation can be made at any time of a transaction execution, and even several times. This validation is a routine to validate the data set, checking if it has been modified by any other transaction. The result of this varies depending on the TM implementation.
- Detected on commit validates the data-set before committing, checking if any data has been modified by other transaction.

Early detection avoids lost computation on a transaction, since it could abort immediately. However there are cases where an early detection aborts the transaction that could have committed otherwise.

Keeping track of object accesses through a list of transactions accessing the object can be difficult to maintain since there are no guarantees of how many transactions will access the data, obligating therefore to an implementation of dynamic structures so that no boundary exceptions occur. This boundary problem may occur with versioning data-sets, since it can as well outbound the max value, but it is a fairly easier problem to overcome.

Several possibilities can happen with conflicts. If a system desires to allow concurrent reads, can even not track the reads like Invisible read TM system [LR06], but any transaction that performed a read on some data-set must validate it before committing.

Scott [Sco06] identified 4 policies for detecting conflicts:

- Lazy invalidation — T1 writes an object. T2 reads the same object and T1 commits before T2
- Eager Write-Read — T1 writes an object. T2 reads the same object but none of the transactions has committed yet
- Mixed Invalidation — T1 writes an object. T2 reads the same object and then writes into the same object and none of the transaction has committed yet
- Eager Invalidation — T2 reads an object. T1 writes into the same object and none of the transaction has committed.

When a conflict occurs, in a Late Concurrency Control approach, a transaction might be executing in a inconsistent state. This has to be either prevented or corrected.

- The validation method requires validation of the read-set to assure that the data is consistent. Depending if "operating" in deferred-update or direct-update mode, rules have to be made to guarantee the consistency of data.
- Invalidation method invalidates the read-sets of an object/data when that data-set is accessed by any transaction aiming to modify the same data-set.
- It is still possible to allow inconsistency toleration, since validation might be expensive. This approach needs to guarantee that none of the other concurrent transactions will execute correctly using any data manipulated by this inconsistent transaction.

Weak and Strong Isolation

There are two types of isolation, weak and strong [BLM05]. When we access data in a transaction environment, we use a STM engine, therefore we expect the engine to protect our data from concurrent transactions. But what if a non-transactional access is made to the data? Weak isolation allows the data access, therefore expecting the programmer not to access data outside of the transaction environment, or access it in a controlled way, so that it does not corrupt the data.

In closed memory systems such as C where malloc()/free() instructions prevail we have to take in account recyclable data, so that non-transactional instructions might access it.

Strong Isolation states that data is protected from non-transactional access, so it is only possible to access it through the STM engine. Basically it turns every single instruction out of Transactional environment into a transaction, forcing it to respect the transaction engine rules.

Transaction granularity

Transaction Granularity refers to the amount of data that is being controlled by the STM for conflicts. Granularity can differ from system to system, some refer to an object, some for blocks of data or words. These are distinguished by it's size, therefore implying fine grain or coarse grain. The finer the grain, more concurrency is allowed, the coarser the grain, more simplified approach and less overhead is achieved.

Coarse object granularity is appellative, since we can associate metadata easily to the object, but if we are accessing just a small amount of data in the object, and other transaction tries to access a different data in the object, it won't be able to.

Blocking vs Non-Blocking synchronization

Early TM systems used nonblocking synchronization, this type of synchronization offers a stronger guarantee of forward progress, while blocking synchronization imposes concurrency limitations.

Ennals [Enn05] suggested that system deadlocks prevention are the only compelling reason to implement non-blocking transactions systems.

TL2 [DSS06] article suggests that experience until now reveal that blocking synchronization STM systems perform better and are easier to implement than non-blocking synchronization systems.

There are mainly three types of forward progress guarantee [LR06]:

- Wait freedom is the strongest guarantee that all threads accessing a common set of objects make forward progress.

- Lock freedom assures that at least one thread makes forward progress from all the set of concurrent threads.

- Obstruction freedom is in others words an optimistic concurrency control, since the only requirement is that a subset of data (partially completed operation) can be rolled back.

Direct and deferred update

Direct and deferred update state how data should be handled when its modified.

Using direct update approach implies that the data is directly updated when it is modified, therefore adding the need to keep a log change, so if the transaction aborts, we are able to rollback all the instructions, leaving a consistent and unchanged state of the data regarding the beginning of the transaction. The data has to stay the same, as if the transaction has never happened.

When deferred update is used, the data is handled locally, through a copy. If the transaction commits successfully, the data then is moved to the proper location. Of course this is not as simple as this description, some considerations have to be taken into account regarding data accessed by other transactions. If no other transaction modified the state of the data, then the effective changes are made into memory. Aborting a transaction in deferred update is as simple as deleting the private copy of the transaction.

The direct update mode speeds up the reads since they simply access shared memory to read, however they maintain locks on variables until transactions commit. On the other hand deferred update minimizes the overall contention by only acquiring the locks on commit time, but the drawback is the necessity to check the write set of current transaction, verifying if any change was made to the variable that we wish to read [Cun07].

Lock Placement

When a update to memory segment has to be made, independently of its granularity, there is the need to lock any accesses to that segment from other transactions. We can opt for defining a variable adjacent to data which will indicate if the data is locked or not, through a single bit variable. There is also the possibility to define a separate table with all the locks.

2.3 Consistent Transaction Layer (CTL)

2.3.1 Introduction

CTL [Cun07, CLD08] was developed in *Universidade Nova de Lisboa* by Gonçalo Cunha as his Master Thesis and adds some functionalities to Transactional Locking 2 [DSS06], a global versioning clock based STM engine presented by *Sun Microsystems*. Some of the added functionalities are the basis of a STM engine design, others performance related. Besides his objective to improve the STM engine, the objective was that CTL would ran in x86 processor instead of SPARC, and that it would use GCC instead of SUNPRO C compiler.

Transactional Locking 2 (TL2) was chosen due to its singular characteristics. Although it is a lock based implementation, it helds locks for very short periods of time (only on commit time), and it suits open memory programming languages such as C. Also, the global versioning clock presented a new approach towards consistency validation of transaction states. TL2 used a redo logging strategy and allowed lock to be placed on a separate table or adjacent to data, depending on compile time options from part of the user. CTL ended as a prototype with more features than the TL2, that will be described in the following sections.

2.3.2 CTL Features

User abort

CTL provides a way to a user to explicitly abort a transaction. This is very useful in some cases, since it might be needed to abort a transaction, not because of transaction inconsistency but due to logic in program execution.

Automatic transaction retry

Automatic transaction retry happens when two transactions collide, therefore one of them has to abort. If there is no automatic transaction retry the programmer has the need to explicitly test if the transaction committed successfully. With Automatic transaction retry implemented in CTL the transaction automatically retries, corresponding to a *exactly one successful commit*. This has been achieved with *setjmp/longjmp* instructions. The *setjmp* routine saves the processor state on the beginning of a transaction, while *longjmp* restores the processor state that has been previously saved by *setjmp*.

Transaction Nesting

CTL transaction nesting is partially based on Closed Nesting. If the execution of a sub-transaction commits successfully the transaction log is concatenated with the log of the outer transaction. In case where the sub-transaction aborts the log is discarded and the control passes to the parent transaction. However when a collision with other transaction is detected even the parent transaction aborts, since the conflicting variable might as well been read by the parent transaction. This can be avoided by validating the whole read set of all transactions, assuring that the parent transaction is still in a consistent state.

Update techniques and validation modes

CTL was developed based on TL2, which used the *deferred update* technique with a *redo logging*. CTL not only implements the deferred update mode, as well as *direct update* mode using a *undo log*. These extra features developed in CTL allows the user of CTL to use, according to application necessity, each one of the approaches. With the deferred update technique, none of the changes is made visible to all other transactions until the transaction commits successfully. This is achieved through keeping a temporary copy of the real values (the redo log), combined with the global versioning clock to assure state consistency of the transactions.

The direct update mode approach, results in direct changes of the memory values, keeping an undo log, allowing rollback in case the transaction has to abort leaving therefore the data consistent.

When a transaction starts, it reads the global version clock into a transaction timestamp. When a transaction loads, transaction checks if the variable is already in the write set, if it does mean that it has already been accessed within the transaction and returns the value written in the write-set, otherwise, logs the read in the read-set and returns the value. For the read to be valid, the transaction must check if the lock isn't held, if the lock version is the same in both checks (before and after the read) and that the lock version is lower or equal to the transaction timestamp, guaranteeing that the transaction is in a consistent state. When a Transactional Store happens, the transaction logs the write on the write set. When a transaction commits, the corresponding locks are obtained, the read set is validated, the global version clock is incremented, the new variable values are copied from the redo log to the corresponding memory positions and finally the locks are released with an updated version corresponding to the new global version clock number. Aborting a transaction is as simple as discarding the redo log (in redo mode).

2.3.3 CTL granularity

CTL allows *Object mode* besides *word mode* [CLD08]. This results in coarser granularity, that can be very useful when several modifications are made in the same object, since with the object mode we only have to acquire one lock for the whole object, reducing therefore the meta-data size, and the transaction overhead of validating all the read-sets in the word mode since only one entry is made to the undo-log for several reads on the same object. However it has also drawbacks, since coarser granularity reduces concurrency.

2.4 Cell Broadband Engine

2.4.1 Overview

CBE began as a challenge to improve the relation between power consumption and performance to high demanding processing applications. Sony, Toshiba and IBM started a joint venture to design an architecture capable of providing such demanding goal.

To be able to take a look into the Cell advantages and disadvantages it is important to talk about metrics. Several metrics are usually used to prove/test architectural design efficiency, such as performance per cycle or design frequency, but as we move into a world where power is more expensive and specially limited, we have to take into consideration the power efficiency of our systems. Several metrics can be used to measure this efficiency, such as energy per operation or performance per transistor. In the design of CBEA these kind of metrics were taken in consideration.

Most architectures now-a-days do not take into consideration the power/performance ratio, and some architectural choices like virtual memory, caches, out-of-order executions and hardware branch predictors decrease this ratio substantially [Pet05], so in CBE, some sacrifices were made in order to improve this ratio. Most of the disadvantages resulting of these "sacrifices" go to the programmer, that have to take some issues in consideration on the development of applications.

Cell Architecture Design

CBEA was finally designed as shown in figure 2.1 and it is composed by a main processor(PPE), eight cores (SPE), an interconnection bus (EIB) and system memory.

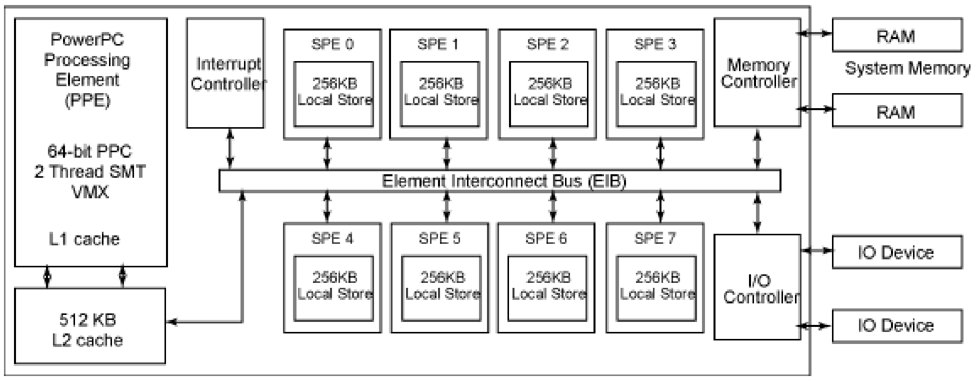

Figure 2.1: General Architecture Design of CBEA.

The CBE has a heterogeneous multi-core processor configuration, pursuing the objective of taking advantage of the parallelism techniques, residing its computational power on the SPEs units. This choice of heterogeneous multi-core architecture was made due to the fact that the architecture would have to handle heavy computation (and therefore the SIMD dataflow of

SPEs are a good choice) and manage an Operating System (OS), that have very frequent context switches.

The Power/Performance ratio was roughly taken in consideration and several choices were made in order to improve this ratio. For instance, the PPE and the SPEs do not support out-of order execution, neither have branch predictors by hardware. These choices reduce the price and power consumption of Cell.

Power Processing Element

The Power Processing Element (PPE) is the responsible for the management of the system. It controls all the 8 cores (SPEs).

The PPE is a 64 bit PowerPC 970, running at 3.2GHz, 2-way simultaneous multithreading processor. It consists of a Power Processing Unit (PPU), and a 512KB 8-way set associative write-back cache, used for both instructions and data as shown in figure 2.2(extracted from *Cell Broadband Engine Handbook. [IBM07a]*) .

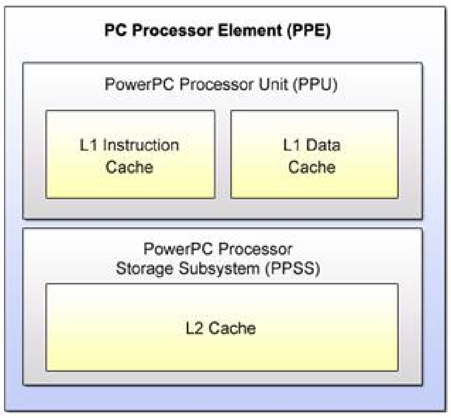
PC Processor Element (PPE)
PowerPC Processor Unit (PPU)
L1 Instruction Cache
L1 Data Cache
PowerPC Processor Storage Subsystem (PPSS)
L2 Cache

Figure 2.2: Power Processing Element General Overview.

Notice that although it is a PowerPC 970 based processor, and it uses its instruction set, its design is much more simpler in order to lower the cost of the hardware. The fact that it uses the same instruction set allows the programmers to start with software based on traditional PowerPC architecture, improving afterwards the software for the CBEA.

The PPU has a 32KB 2-way set associative reload-on-error instruction cache and a 32KB 4-way set associative write-through data cache.

Synergistic Processing Element

The Synergistic Processing Elements (SPE) are 128 bit SIMD ”slaves” of the SPE, and the true potential of computation workload of the CBEA. A single SPE contains a Local Store (LS), Synergistic Processing Unit (SPU) and a Memory Flow Controller (MFC). The LS consists of 256KB

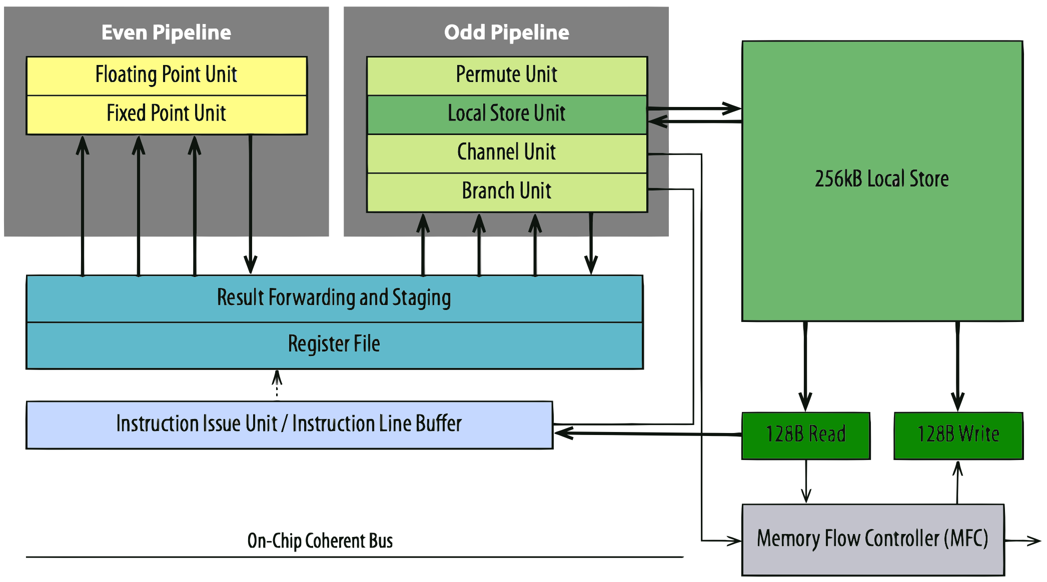

Figure 2.3: Synergistic Processing Element

of unified memory for both data and instructions. The only way to move data to and from LS is through the MFC, which is responsible for all data transfer capabilities of the SPE.

The SPE uses a three level memory organization, (Registers, Local Store and System Memory), that breaks from conventional architectures. This three level organization, combined with asynchronous DMA transfers explicitly parallelizes both computation and data transfers. With this approach, it is possible to minimize the latency on accessing system memory on conventional systems.

In figure 2.3 (extracted from *SCOP3: A Rough Guide to Scientific Computing On the PlayStation 3*) we can take a deeper look at the SPE. We can see the interaction between LS and MFC for data transfer, as well as the two SPU pipelines, the odd pipeline and the even pipeline. The odd pipeline is responsible for memory operations, while the even pipeline is responsible for the computation. This pipelining allows us to perform two SIMD instructions per cycle, one compute instruction and one memory operation.

2.4.2 Memory and Communication

Memory in CBE

Memory in CELL, due to its heterogeneous multiprocessor architectural design, is decentralized. This introduces a very sensitive area of CELL since taking good advantage of the architecture depends on the effective usage of the memory.

The way the PPE and the SPEs access main memory differ substantially. To access the main memory, the PPE accesses main storage with load and store instructions, to or from a private register file. The SPE's on other hand must move data from main memory to its Local Store

(LS) before the SPU can fetch instructions.

Since the SPE's can only execute code inside its LS, data has to be moved into it from main memory and then moved back to main memory. This is possible by using DMA transfers. This data-flow from SPE's LS and main memory is intermediated by two interface controllers, the MIC (Memory interface controller) connected to the system memory and the MFC (Memory Flow Controller), in which the DMA engine is integrated. The MFC allows the SPE to pipeline the data while working on current computation, therefore providing a very powerful combination for non-locking computation. These two interfaces provide several mechanisms to the programmer to manage memory and synchronization.

The Memory Flow Controller provides to the SPE data transfer and synchronization capabilities. It implements the interface between SPU and EIB, LS and system memory. Notice that any data transfer can be initiated by either the SPU or SPE.

The Cell BE memory and DMA architecture are fairly flexible. The PPE provides to the MFC some resources, such as MMIO registers, with effective-address alias on main storage, so others SPEs or the PPE can access and control the SPU. The local stores of the SPEs are mapped into the global address space. The PPE can access (through DMA) local stores on the SPEs, and can set access rights. The SPEs can initiate DMA to any global address, including the local stores of other SPEs.

Communication

There are three possible ways of communication in CBE. Through DMA transfers, signals or mailboxes. The most important one is obviously DMA transfers specially due to its non-blocking facility.

DMA are non-blocking transfers, which allows to pipeline the data while computing the data. DMA can be issued from PPE to SPE, from SPE to PPE and from SPE to SPE. DMA messages are not ordered and they have a maximum size of 16KB, but it is always possible to combine multiple DMA transfers to move data bigger than 16KB. DMA lists are perfect for this, and they can combine up to 2048 DMA transfers. Each SPE is capable of queuing 16 requests. Also there is a proxy queue, that the PPE and other SPEs are able to access. This proxy queue can queue 8 requests. These lists are made available by the DMA engine in each MFC.

Signals are 32bit registers in SPU. Each SPU have two signal-notification channels. They are a very trivial way of communication, it can be used to signal the completion of a task. Each register has a corresponding MMIO register on which the data is written by the sending processor. Each SPE can only access his own registers, therefore if any other SPE, or even the PPE, has the need to access the information, it can be done by accessing the respective MMIO register. Any read made by the SPE on its own registers will clear atomically the channel, contrary to any MMIO read, that wont clear the channel.

Mailboxes are queues for exchanging messages. Mailboxes are blocking operations and some precaution is advised to use it, since any request to read on a mailbox that is empty will block the SPE. The same behavior happens when you write in a full mailbox. There are counters available to verify the state of the queue. Still, they provide a good way to message another SPEs. Mailboxes are 32-bit messages, and they have a FIFO queue. Each SPE have a four-entry inbound mailbox and a two one-entry outbound mailbox. Mailboxes are not a good way to acknowledge transference completion because SPE knowledge of DMA transfer completion is that the local buffer is ready for reuse, so there is a possibility that the mailbox will be read by the PPE but the data isn't there yet. Also, obligating the PPE to continuously check the outbound mailbox of the SPE might flood the bus.

Barriers and Fences are mechanisms to order DMA messages within the queue or tag group. Issuing a barrier will order it with respect to messages issued before and after the current message. A fence will only order it to respect of the previous messages. Although these mechanisms are very useful, it is recommended to use them as less as possible since it wont allow the arbiter to improve the performance of the DMA transfers.

2.4.3 Programming Models

This section exploits the programming models for CBEA. Achieving the best performance on any architecture depends on the exploitation of its specifications. In CBEA we can define two main models, PPE-centric, and SPE-centric.

As CBE attracts several different sectors of computer science community, there are several programming models that fit well in this architecture [BLKD07].

Function-Offload Model

Also called Remote Procedure Call Model, this model is PPE-centric. The PPE manages the application, and offloads the computation to the SPEs. It is the most basic approach to the CBEA. After identifying the code to be run on the SPE, a remote procedure is called through a stub, hiding the communication issues to the programmer. This stub will manage the procedure and communication both ways.

Computation-Acceleration Model

This model is SPE-centric. It uses the PPE almost only as a system service or a controller. This model relies on the SPE for high computation, so it is more adequate for computational demanding tasks.

It can use shared memory among the SPEs or a message-passing model for data bulk movements. The workload can be partitioned either by the programmer or the compiler. Shared and distributed techniques are used in this programming model.

Streaming Model

The streaming model treats the SPEs as pipelines, where each SPE is responsible for a different task (not necessarily though) on the data. The pipelining can be serial or parallel, being possible to achieve better results in a two data streams model than in a one data stream model as stated in [JGMR07].

This approach is suited for computation demanding tasks and also for the tasks that have the need for big transfers of data, since the internal bandwidth available between the SPEs is from far much bigger than system storage. But notice that it is important that the workload can be divided in equal parts, otherwise, there is no sense in having an SPE with a huge workload and another idling. In this model the PPE simply acts as a stream controller.

Shared-Memory Multiprocessor Model

The CBE can be used as a shared-memory model. With DMA cache coherency we can implement this model by DMA commands, from LS to the shared memory and vice-versa. The PPE and SPE have all the same address space in memory. This is only possible because of the global addressing schema of the CBE. DMA also provides lock-line commands, which allows us to make atomic update primitives.

Asymmetric-Thread Model

The Asymmetric-Thread Model is very widespread on SMP. In CBE this Model is possible, and is even implemented by the *SPU Runtime Management Library* SDK, but the preemptive context-switching in the SPEs impose some overhead and costs, since SPEs support it more for debugging purposes.

User-Mode Thread Model

The User-Mode thread is a model where a thread is run by an SPE, which manages *microthreads* or *user-threads*. The SPE thread is supported by the Operating System, but the *microthreads* are supported by user-software without the interference of the OS.

It is possible for the microthreads to be ran in different SPEs. The SPU schedules task in shared memory, that can be processed by any SPU available.

This model has the advantage that has a predictable overhead, since it is running on a set of SPUs, managed by a SPE.

2.5 x86 vs Cell Broadband Engine instruction set architecture

Instruction Set Architecture (ISA) is the boundary between software and hardware. In other words it allows privileged software to manage the hardware. The ISA defines the instruction set (range of instructions available), datatypes available, the addressing modes(define how to calculate the effective memory address of an operand by using information held in registers) and the instruction formats.

x86 is a Complex Instruction Set Computer (CISC) while CBE is a Reduced Instruction Set Computer (RISC). The fact that x86 is a CISC is more due to retro-compatibility than any other reason.

There are several types of addressing modes, they specify how to reach the operands. The instructions addresses specify how to calculate the effective memory address of an operand by using information held in registers. There are several different types of addressing modes like *Register Direct* or *Register Indirect*. The first define the value of operand directly in the register, the former define the address of the operand in the register.

2.5.1 x86

The x86 architecture is a variable instruction length, primarily two-address CISC design with emphasis on backward compatibility. x86 allows non-aligned data, but it is not advisable to do so, due to performance issues. The x86 provide 8 *General Purpose Registers* (GPRs), six segment registers, one flags register and an instruction pointer. x86 is little-endian meaning that multi-byte values are written least significant byte first.

x86 supports 32bit 16bit and 8bit datatypes, single and double precision IEEE floating point.

The operand types in x86 can be passed mainly in 3 ways, directly in the instruction, it can be stored in register and it can be in memory. Obviously if the operand is stored in memory a bus access has to be made, therefore making the execution slower. The reduced number of GPRs has made register-relative addressing (using small immediate offsets) an important method of accessing operands.

Several extensions to the ISA have been added to x86, like MMX and Streaming SIMD Extension (SSE) which allow SIMD instructions and advanced math operations.

2.5.2 CBEA

Cell Broadband Engine is as we know a heterogeneous multiprocessor architecture intended to support a wide variety of needs. As SPEs are designed for computationally intensive tasks and therefore SIMD processors, their Instruction Set differ from the PPE, actually being very close to the PPE's Vector/SIMD Multimedia Extension (VMX or AltiVec). This means that we have two different instruction sets in CBEA. The instruction set for the PPE, which is an extended version of the PowerPC instruction set, and the SPU ISA.

Power Processing Element and PowerPC

The PPE instruction set is based in the PowerPC 2.0.2 ISA and has some enhancements. This enhancements are the VMX, that add 128 bit datatypes for vector and scalar operations, some new instructions and C/C++ *Intrinsics* for VMX. *Intrinsics* are C language commands, in the form of function calls, that can substitute one or more in-line assembly-language instructions. The *CBE programming handbook* [IBM07b] gives a general view of the PPE ISA but detailed descriptions can be found in the *PowerPC user Instruction Set Architecture* [WSMF03] manual. CBE provide 32 General Purpose Registers (GPRs) [IBM07c].

The data types supported by the PPE can be viewed in table 2.1.

In CBE when a instruction is presented to the processor, the two most low-order bytes are ignored. The addresses point to the *Most Significant Byte* (MSB) since CBE uses the big-endian

Table 2.1: PPE supported datatypes

Datatypes	**Lenght**
Fixed Point	
Byte (Signed and Unsigned)	8
Half word (Signed and Unsigned)	16
Word (Signed and Unsigned)	32
Double Word (Signed and Unsigned)	64
Floating Point	
Single precision	32
Double precision	64

convention.

The load and store addressing modes always define a base index and there are three different possibilities:

1. *Register* - Load or store the contents of *Register*

2. *Register + Register* - Indexed form of the Load and Store instruction form the sum of the contents of *Register* plus the contents of base register.

3. *Register + Displacement* - Forms the sum of Register and a 16bit signed-extended immediate field of instruction and sums the content of base register.

Instructions in PowerPC are 4 bytes long and aligned on word (4-bytes) boundary. It can have up to three operands, most of them specify a 2 source operands and one destination operand. In table 2.2 we can see those instructions. Below we introduce the instruction types available for PowerPC.

Synergistic Processing Unit

The SPU ISA provides 7-bit register operand specifiers to directly address 128 registers using a SIMD computation approach for both scalar and vector data. The SPU ISA operates on SIMD vector operands, with support for some scalar operands. Any scalar operand must be issued within the preferred slot, that are the left-most bytes in the registers (the 4 first bytes). Notice that SPU is a Load/Store architecture, which means it can only access data to move it to the registers, in other words, it cannot operate in LS.

The datatypes supported by the SPU are: byte, halfword, word, double word and quadword as shown in the figure 2.4 (extracted from *Cell Broadband Engine Programmer Handbook*).

The instructions in SPU have to take in consideration the existence of two pipelines. This allows two up to two instructions per cycle. The pipeline destination depends on the type of instruction issued.

The C-language *intrinsics* set [IBM08a] represent in-line assembly-language intrinsics in the form of function calls. They provide the programmer with explicit control of the SPE SIMD

Table 2.2: Types of Instructions in PowerPC

Type of instruction	Description
Load and Store	Fixed point or floating point loads and stores. The fixed point loads support byte, half-word, word and double word loads and stores between storage and GPRs. Floating point supports word and double word load and stores between storage and floating-point registers (FPRs)
Fixed-Point Instructions	Arithmetic, compare, logical and rotate/shift instructions
Floating-Point Instructions	Floating point arithmetic, multiply-add, compare and move instructions
Memory Synchronization Instructions	They control memory operation order. They include load and stores with reservation features.
Flow Control Instructions	Instructions flow control mechanisms like branching
Processor Control Instructions	Synchronization for memory accesses and cache managing capabilities.
Memory and Cache Control Instructions	These control caches, TLBs and segment registers

instructions without directly managing the registers. It is also in the interest of the programmer to use this extensions since the compilers that supports them will produce efficient code for SPEs. Intrinsics allows the user to make not only data *Load and Stores* and instruction scheduling but much more.

Intrinsics are divided into three sub-areas. *Specific intrinsics* are the functions that are mapped one-to-one with assembly in-line instructions. *Generic Intrinsics* are the functions that include more than one assembly in-line instruction. *Composite Intrinsics* are the most complex type, they can be considered as a list of *Generic Intrinsics*.

Since the LS is single-ported (can only access one address per cycle) and it is a unified memory for both instructions and data, it might happen instruction starvation, since DMA transfers have priority over instructions fetches. To try to avoid this, it is useful to transfer as much data as possible in one chunk, for both DMA transfers and instruction fetches. The LS allows 32 instruction fetch loads per requests.

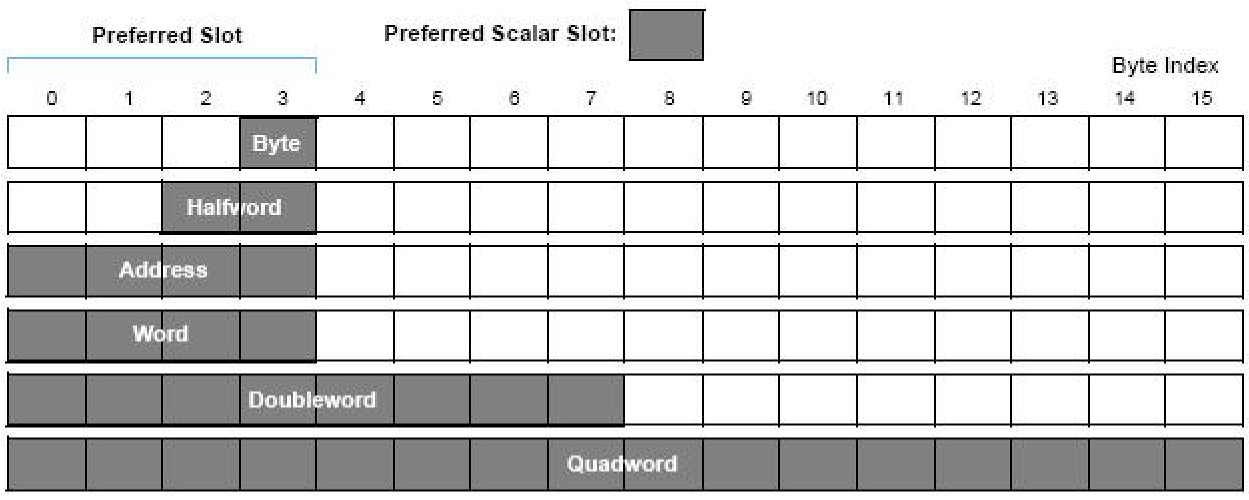

Figure 2.4: SPU registers, data types and preferred slot.

2.6 Software Managed Cache

Caches are storage mechanisms that duplicate data in order to increase performance, that otherwise would have to access some higher latency cost memory/storage. Widely known in form of hardware caches on modern CPU's, they store data in small hardware mechanisms part of CPU in order to improve performance and reduce accesses to system memory. The concerns in caches has always been managing inconsistency on several private copies, since data structures or variables are replicated through the caches. Therefore this inconsistency has to be controlled, either by software schemes or hardware schemes. Caches are used on systems to improve the performance of irregular pattern in memory accesses since they have a good probability of a given data is already present "locally" instead of having the necessity of accessing system memory. This happens by exploiting several techniques like the spatial locality of memory accesses.

DMA transfers can be of high cost depending on the granularity of the data being transferred. One way to improve this is to make DMA transfers as big as possible in order to reduce the overhead. In the approach where user fetches data from main memory "on-demand", that is, as the data is being requested, it might happen that the buffer (EIB) will be filled very quickly, reducing performance.

For the reasons stated above a private software-managed cache can be used in SPE's LS in order to improve performance. Specially regarding the write-set, since each load in the scope of a transaction must verify the write-set for previous changes on that address. Since we will be flushing the write-set into main memory, due to memory limitations of the LS, the latency will increase upon a `TxLoad()` command, therefore, using a cache for the write-set will hopefully reduce the latency on accessing the write-set.

In this particular case, CBE architecture, the LS can be regarded as caches, where they fetch data from main memory. This way it is expected that a *software managed cache* can improve the hit-ratio of data on LS and reduce the overhead of DMA transfers, since transfers will fetch not only the specific data but also the surroundings.

CBEA provides a *software managed cache* as an API for usage on the SPEs. This *software managed cache* combined with the STM framework can provide a powerful tool towards performance and efficiency.

3 Architecture solution

3.1 Introduction

Transactions allows to execute pieces of code atomically, which results in either all the operations inside the scope of the transaction being executed return successfully or none will. Most modern processors architectures are evolving to multi-core. To really take advantage of multi-core or multi-processor architectures it is necessary to rethink the whole applications in order to take advantage of the parallel programming techniques. Therefore STM has been gaining popularity and importance by taking advantage of this multi-core architecture effectively instead of simply dividing threads into each processor core.

In such architectures, like CBEA, it is expected that concurrency and parallelism techniques improve performance substantially, however the high performance solutions presented for CBEA are very specific due to its novel architecture and memory distribution and it is still hard to develop tools that are able to provide an abstraction layer to the programmer that is able to exploit concurrency and manage consistency. Software Transactional Memory is a programming model that proposes this abstraction layer, and is gaining increased popularity and several prototypes have been developed with performance close to fine-grain specific implementations for the domain problem. The possibility of using STM to develop a tool capable of hiding all the memory management and consistency in CBEA is very appellative. In this chapter we propose to specify a STM engine to CBEA and discuss the advantages and disadvantages of different STM approaches on implementing such engine.

Developing a STM framework to CBE presents several challenges due to the unique design of this architecture. Most of the known STM frameworks execute in the address space of a single computer, which means the processor is able to load/store directly from system memory/registers. This is similar to CBEA if we take in consideration only the PPE, but not if we want to use the whole set of processors available (PPE plus SPEs), since SPEs are not able to directly access system memory. This introduces the challenge of effectively using the three level (registers, local store and system memory) SPEs memory that breaks from conventional architectures. The usage of a previous STM framework as a basis, Consistent Transactional Layer (CTL), is made due to its feature richness and good performance achieved, namely the redo and undo log modes and automatic transaction retry in case of failure.

3.2 Challenges

In an architecture where the sub-processors do not allow direct addressing of system memory, instead using DMA transfers for data transfer among main and local memories, imposes higher concerns when developing a simple program. This happens because the data that is being moved between the main memory and the sub-processors have no integrity check, being possible that different copies of the same data are manipulated concurrently, therefore delegating the concern of integrity check to the programmer of the application.

We will now discuss the overall problems in programming in such peculiar architecture. This represent general issues. Afterwards the specific problems that an implementation of a STM framework might bring are approached.

In CBEA, the LS's size limitation of 256KB for both code and data is really the main concern. This can be fulfilled very quickly and some techniques like code overlay [IBM07d] are used in order to overcome the problem for the code size problem. For the data bulk movement problem the correct management from the programmer is expected in order not to exceed the size allowed.

Mailboxes are 32 bit messages and are used as communication mechanisms between PPE and SPES and also between SPES. As they can be a good mechanism of communication they may also impose stalling of the processor, since reading an empty mailbox will stall until an entry is detected. The same happens when writing into a full mailbox. This might even be the expected behavior of the program, since execution may be unfeasible until the expected message arrives. But this stalling behavior in a non-deterministic execution flow of concurrent threads might be a problem.

So when it comes to develop a STM framework capable of using SPE's in a transactional environment its possible to identify some problems.

1. No matter we are working in deferred update mode or direct update mode, a log has to be kept for each transactional read or write. This mechanism, combined with the low space available on SPE, imposes even more space problems. The STM future user, which is unaware of the details of the implementation will face even less memory available if we choose to follow the natural solution of keeping the logs locally in LS. These might grow very fast, and that will depend on the specific program using the STM.

2. When working in an SPE perspective, we imagine we need to make a change, actually a very sensitive change in some remote memory location (main memory). In these cases

an atomic CAS is most of the times made (in form of inline gcc asm), in order to prevent any preemption from the cpu. Atomic functionality through DMA can be made from the SPE side to main memory, making a correspondence to the *lwarx* and *stwcx* (processor atomic instructions) PPE operations. Still this represents a higher latency if we wish to implement a routine which has several steps of verification (Lock acquisition, effective changes on memory positions and completion of validation).

3. DMA transfers must be aligned to boundaries. Maximum efficiency is reached when memory is aligned to 128 bits, being possible to transfer 1, 2, 4, 8 and multiples of 16 bytes. So it is not possible to transfer a structure that has 6 bytes, unless we divide it in two distinct transfers or we pad the structure to one of the possible transfer sizes. This concerns the development of the STM framework but also the user that must take this in consideration when coding for CBEA.

3.3 Solution Layout

The CBE, considering it is a Heterogeneous multi-processor architecture with local store for each SPE, intuitively fits well in a *Transactional Memory* framework working in deferred update mode, since we would compute all instructions on the SPE's and we would only need to validate the read/write-set on commit time reducing overheads on accessing main memory. This goes toward a function off-load model, where we offload task to SPE's to be executed. The costs from working on direct-update mode would grow, since we would increase latency on accessing main memory for each write to a variable, therefore this approach is at first sight undesirable. Minimizing accesses to main memory equals minimizing latency resulting in an increased performance.

The proposal is to use a shared-memory model, where transactions are executed in the SPEs and the PPE simply acts as a manager and transaction validator. Although transactions are allowed in the Power Processing Element (PPE) the focus will be on developing the support for transactions in the SPE's due to their high processing capacity.

In this approach the SPE fetches data necessary to its own LS (on demand), computing the code and changes are only made visible upon commit, meaning we will be working in a deferred update mode. This represents developing a library to the SPU that will allow the programmer to start a transaction within SPE context, load a variable, either from main memory or local store, store a value and commit a transaction. On the other side, the PPE, we will have also a library (an extended CTL) which will be able to validate the transaction.

3.3.1 Design Choices

As stated above, a deferred update *STM framework* seems to fit better in this particular architecture than a direct update *STM framework*.

In order to develop a STM framework to CBEA and study the impact of the design choices in performance a base model is presented and afterwards a multi-bufferd model.

1. As a first model approach (Fully Local Model) a simple framework is capable of executing transactions in SPE environment. This straightforward approach intends to present a functional way to execute transactions without any special performance improvement nor memory size concerns. This model has a limitation which is the memory. In this model the programmer has to be aware that it might happen that a very big transaction might out limit the memory space allowed in the LS. This is because of the write-set and read-set having a fixed size in LS (SPE) and that the log is never flushed into main-memory. This model suits for transactions with low workload and low data bulk movement which will never overpass the LS max size. Although this approach has clear disadvantages concerning memory it has also advantages, since the memory is always present in LS reducing overhead on memory communication to main memory. So if it is guaranteed by the user that the transaction will never exceed LS size it might even be a better approach than the following model. This will be further discussed in Section 3.5. An interesting approach that would complement this one would be to make an intermediate step between first and second model where the read-set is flushed but not the write-set, this would still translate in high performance and it would free some space.

2. The second model (Multi-buffered Model) aims at solving the memory limitation on LS. In this model a multi-buffered approach is implemented in order to flush the local log into main memory. Therefore this model overpasses the memory limitation by using buffers to keep the transaction log. This way whenever one buffer is full, that buffer is swapped with an empty one and the log is flushed into main-memory in order to free more space for the user. With this approach the user will be able to execute big workload transactions. In this model the *Read Log* can be flushed into main memory without any further concern, but the same does not happen with the *Write Log*. According to our model the transaction log is dispatched to main-memory introducing a problem. Whenever a transaction issues a TxStore, which is writing into a variable, it is necessary to verify if that memory position has been written previously in the current transaction scope. Therefore any TxStore that does not find an entry in local memory, for the specific address being written, can not assume that it does not exist, it must verify main memory to check for the log entry. This is made by searching remotely in the main memory for the given address and returning a value.

In table 3.1 we can see the properties of the Model. Our Model provides a weak isolation since we allow non-transactional accesses in transactional scope. We use word granularity in our framework working in deferred update. The synchronization method is blocking on commit time, since in order to effectively issue changes in memory we need to assure no other transaction will corrupt the data, therefore using a blocking mechanism to protect the data-set from unwanted changes. Also we provide an early inconsistency detection mechanism in our algorithm. Regarding transaction nesting, it is only supported in PPE transaction scope.

Table 3.1: STM solution properties

Property	Value
Isolation (Weak or Strong)	Weak
Transaction Granularity	Word
Direct or Deferred Update	Deferred
Concurrently Control	Versioning
Synchronization	Blocking on Commit time
Conflict Detection	Early inconsistency detection or at Commit time
Inconsistent Reads	None
Conflict Resolution	Abort Conflicting transaction (abort backoff exponential)
Nested Transactions	Closed Transaction Nesting on PPE environment
Exceptions	None

3.3.2 Algorithm

In this section we present the algorithm for the execution of the STM framework. First we introduce the fully local model as described in Section 3.3 along with figure 3.1. Afterwards the multi-buffered model is presented supported by figure 3.2.

Fully Local Operational Model

As suggested by its name, this model assumes a fully local transactional execution that will not exceed the LS size. It is responsibility of the programmer to not exceed this limit. Note that the logs are protected against overflows but the correct execution will not behave as expected if LS is full since it will be impossible to issue Transactional Loads or Stores. This model presents a simple approach towards developing a STM framework. Figure 3.1 illustrates the behavior of the execution and its corresponding data movement.

Lets analyze in detail how the transaction behaves in Fully Local model.

- A transaction is initiated on PPE side. All the necessary data structures are initialized in main memory. The SPE execution is started and the memory addresses are passed as arguments to the SPE thread execution.
- As the execution starts on SPE side, the memory references which are passed by argument are obtained on a call to `TxStart()`. These includes both the STM framework and the user code memory references.

  This is due to the design choice of passing the library memory locations by argument and not by a message passing mechanism. All the process of initializing the framework works in non-transactional context.
- A `TxStart()` is issued in SPE context. This starts all the local transaction data structures, namely the redo log and global transaction information.
- Whenever a `TxLoad()` is issued, the correspondent data is searched in LS, verifying if it is already present locally. If it is, the correspondent value is returned to the user, otherwise

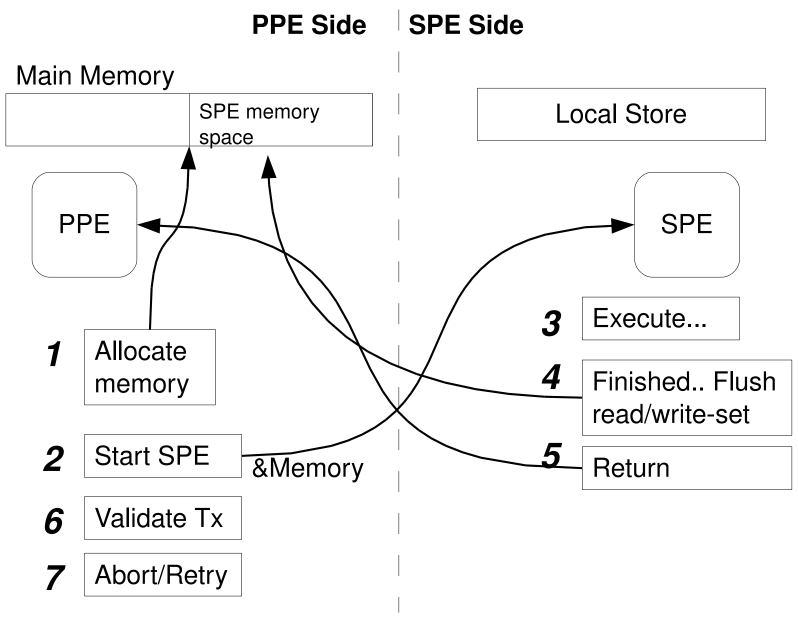

Figure 3.1: Visual caption of Algorithm for Fully Local Model.

the correspondent data-set is transferred from main memory to a local store address and the data value is returned to the user.

- A `TxStore()` will simply record the value in the redo log.
- A `TxCommit()` command will start the validation step. The log is transferred to main memory and the SPE transaction is terminated. The PPE will validate the read-set in order to issue the write-set changes. If the read-set is consistent then the transaction commits, otherwise it is retried.
- If a `TxAbort` occurs, the SPE transaction is terminated and the all the local data structures are freed.

Multi-buffered Operational Model

The Multi-buffered model aims to overpass the memory limitation of the local memories in SPE. By keeping the logs in buffers we can dispatch the buffers into main memory, releasing space in LS for further use. As stated in Section 3.3.1 this model introduces the problem of a given log entry not being present in local store when a TxLoad is issued, which has the necessity to check for previous entries in current transactional execution scope. This is solved by searching for the given address in the transactional log in main memory. Figure 3.2 illustrates the behavior of the execution and its corresponding data movement.

In step 2 of the figure the variable `&Memory` sent to the SPE refers to the memory address allocated in step 1, which is the library memory space for that specific SPE, meaning that is the correspondent address that the transaction will use to transfer the transaction log.

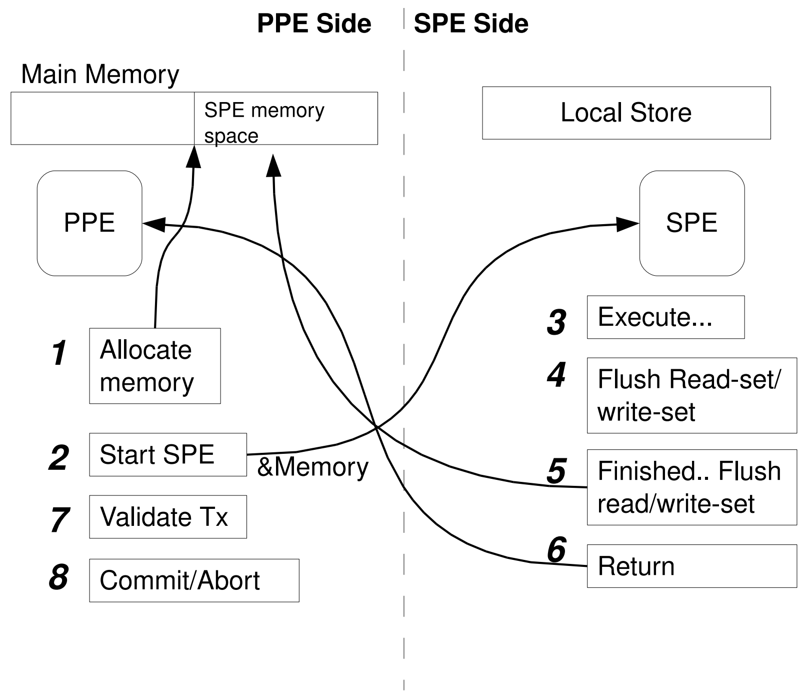

Figure 3.2: Visual caption of Algorithm for Multi-buffered Model.

The step 4 represents the mid-step of flushing the read-set to main memory in order to obtain more free space.

Lets analyze in detail how the transaction behaves in Multi-buffered model.

- A transaction is initiated on PPE side. This results in starting the necessary structures in main memory (Transaction ID), mallocing memory for further transference of data on the end of SPE *Transaction* execution for the PPE side, and afterward, initiate the SPE context, loading the program execution of the SPE side and passing by argument the pointers necessary to the SPE so it knows which memory positions to fetch data from, and where to store it.

  This memory positions could be passed by a message passing mechanism, but these approach is more effective since we don't need to dynamically adapt the memory locations during runtime.

- A Transaction is started within the SPE context. This will create the necessary data structures internally in LS, specifically the write/read-set and store the clock version received by argument.

  The resulting execution on SPE side starts the execution with the following premises. First, it knows the *Effective Address* where to fetch the data necessary to the user program (user space memory). It also knows the *Effective Address* where to load/store STM related data (library space memory). Afterwards the normal execution of the user program will occur.

- When a `TxLoad()` is issued, this will trigger a verification, checking if the data-set requested is already present in the write-set, if it is, the correspondent address is returned and the read-set is updated. If the data is not present locally (in LS) then it is necessary to fetch the data from main memory through DMA transfer, returning the address of its new current storage in LS.

  Here we use a double-buffering approach to maintain the read-set. This results in two buffers. As soon as one of the buffers is full the second buffer is used automatically, while the first is flushed into main memory for validation later at commit time.

  When unavoidably we need to dispatch the write-set into main memory, the SPE is responsable for verifying the write-set present in main memory for the correspondent data. This will impose higher latency, but it is necessary in order to allow, for instance, running big transactions.

- When a user issues a `TxStore` an entry is logged into the write-set, in the form of a pair `AvPair(address,value)`, of the current transaction in order to be committed afterwards. A double-buffer technique is also used to maintain the write-set, since we want to keep as most free space in the LS as possible.

- A transaction commit call (TxCommit) happens on the end of the current scope of execution. At this point it is necessary to validate the read/write-sets in order to effectively commit the transaction and the changes be made visible in main memory. For this to happen the read/write-set are moved into main memory to the previously determined address malloc'ed by the PPE and that was received upon beginning of the transaction. Therefore the DMA call transfering the data is made, and the PPE is responsible for validating the dataset. Since the PPE knows which memory previously allocated corresponds to which SPE, the SPE simply has to halt, waiting on an answer of whether to commit or to abort.

- When a Transaction abort call occurs (TxAbort), the local files are simply discarded and the execution is terminated.

- The PPE validates the read-set. This happens by verifying each entry's version with the current version present in main memory. If any of the main memory entry's version exceeds the version presented in the read-set then the transaction must abort.

In figure 3.2 it is possible to see the behavior of the STM framework during the execution. The numberings represent the temporal execution of the steps. For simplicity we only represent one SPE.

3.3.3 API

This section describes the API available for the programmer.

SPE library – tl_spu

- `unsigned long long TxStart(long long int argp);`
  Starts a transaction, receiving as argument (argp) the composed structure of pointers to user memory address and library memory address (see Section 4.4 for more details), returning the user memory address.
- `void TxLoad(unsigned long long dest, intptr_t volatile address, int size);`
  Loads a value from main memory, located at *address*, of a given size to a given destination.
- `void TxStore(unsigned long long address, intptr_t value);`
  Stores a value into a given address.
- `void TxCommit(void);`
  Commits a transaction.
- `void TxAbort(void);`
  Aborts a transaction.

PPE library – tl_ppu

- `TxStartSPE(spe_program_handle_t spuCode, intptr_t struct);`
  Starts a transaction in SPE executing the given user code in variable spuCode. The second argument is the user struct that is intended to be passed by argument and it can be null.

3.4 Evaluating Design Choices

In this section we intend to describe what are the consequences and advantage of the design choices over our two models, relating performance, memory operations and architecture dependent operations.

3.4.1 Deferred update vs Direct update - Transaction Log Management

Some choices made in the modeling of this framework are directly related to the CBE architecture or similar heterogeneous multiprocessor architecture with this 3-level memory organization. This kind of architecture is unique until now, but the same approach would result in the same benefits in any similar architecture. The fact that we use deferred update instead of direct update relates directly with the fact that the SPE's can not address directly memory, having to DMA any data from memory to their LS. Considering this fact, a direct update approach for a STM framework would have to deal with very high latency each time it would want to make a replacement on memory. This is undesirable since we aim at achieving the best performance possible. Therefore a deferred update mode allows to execute the transactions locally on the SPE's and only committing the changes on the end of execution. This provides a much more viable approach then the direct update mode since it minimizes accesses to main memory, specifically the writes, since we will be recording the redo log locally in LS.

Also this goes towards the *Computation-Acceleration Model* where a transaction is offloaded into a SPE.

Although this approach could be made simply by keeping the logs in SPEs LS, the low size of the LS (256KB) introduces the problem of transactions manipulating high quantity of data. It might easily out limit the LS size which is a problem. To solve this problem, we can transfer the log into main memory to a previous determined location. This way it is possible to free space in the LS. Now in order to do this efficiently we must use a double buffering approach for the log. This way we can transfer one buffer, while using the other, due to the capacity of the SPEs to execute one processor instruction and one memory operation per cycle. Also important to refer is that this movement of data could also be made only for the read-set. Considering the most basic model (see Section 3.3.1) we could keep the write-set always local but the read-set be multi-buffered to main-memory. Since transactions do not need to verify the read-set during execution this can be a good intermediate solution.

This is due to the deferred update approach, since any change during the scope of the transaction is not made visible until successful commit. The changes are kept in the write-set, therefore imposing the verification each time we read a value (TxLoad) if the data has been manipulated during current execution. This verification is made through verifying the write-set for the specific data. So if we transfer the write-set to main memory we will be increasing the overhead of verifying the write-set, leading to the same inconvenience as the direct update mode, higher level of main memory accesses.

Still it would be a huge limitation of the STM framework limiting the number of writes on a scope of a transaction, therefore even though it imposes higher latency accesses it is provided the guarantee that a transaction may make as many writes as necessary. This is made by flushing the write-set into main memory.

The Multi-Buffered model keeps releasing space in LS to main memory and will reach at some point of execution a *Commit Transaction* `(TxCommit)`.

At this point the remaining log is flushed into main-memory and the PPE follows with the validation of the transaction issuing the changes into main memory in successful case or restarting the SPE execution in case of failure.

It could be possible to commit the changes directly through the SPE, with no PPE involvement. This is a design choice in the development of this framework and is discussed in the next section.

3.4.2 PPE vs SPE Validation

Lets analyze the possibilities regarding SPE vs PPE validation.

1. In the full local transaction mode (see Section 3.3.1) the transaction could be validated in the SPE. Assuming we have all the log locally we could verify it, making a set of DMA operations in order to read and write the main-memory library space data, acquiring the locks and issuing the necessary changes into main memory regarding user-space memory. Atomic DMA operations allow us to perform atomic operations into the main memory. Although this has some interesting possibilities it highly increases the cost of validation due to the latency on DMA accesses to main memory. It could be interesting to see the results but our approach goes towards PPE validation. This way latency is reduced and free's the SPE for another transaction while the PPE validates the current transaction. Also it keeps the tracing mechanism implemented for CTL working.

2. The multi-buffered model could also be validated locally in the SPE but it would even impose more latency since some of the logs are already in main memory. The PPE validation scheme requires the transfer of the read-set and write-set to main memory to a previous allocated memory space. This way we only have the need to transfer the rest of the log. After this data transference the execution returns to the PPE that must validate the transaction. The PPE then has all the necessary information in order to validate the data. On end of validation, the PPE then either aborts or retries the transaction.

   Considering both approaches, we believe that validation through the PPE is more reasonable and effective since it reduces memory accesses and it might allow fast release of valuable resources, the SPE, while the PPE validates the transaction.

   Considering the use of a Software Managed Cache could greatly improve performance, since it would (in most cases) reduce the number of main memory accesses by exploiting all the caches properties. Using a cache would improve the probability of a given data (write-set/read-set) being already in LS instead of having to be transferred from main memory. A cache could minimize the effects of checking for a given entry of the write-set in main memory. Also in order to improve performance a bloom filter is used locally in each SPE regarding the remote main memory locations accessed in Transactional scope.

3.4.3 Framework configuration

Some improvements that can be made are related to the configuration of the transaction to be executed in the SPE's. Considering the full model, with double-buffering approach the user should be capable of, considering the given user code, define the size of memory that SPE library should reserve. So intensive demanding tasks with reduced data set can be given more space to execute locally by the user of the library. This will result in a performance advantage since any check on a previous written log will only need to verify locally sparing the latency on a main memory access.

3.5 Conclusions

This approach towards developing a STM framework for CBEA aims at delivering a high performance platform where a programmer might concurrently execute several threads maintaining consistency of data and correctness of execution without big concerns except to identify potential critical areas of code that need to be protected.

Also we present two distinct models in order to better evaluate the possibilities towards developing a STM framework for CBEA. With this architectural solution we intend to prove the usability and also state the versatility of this architecture and several possibilities that it provides to the programmer.

4

Implementation

4.1 Introduction

In this section it is discussed the practical implementation of the provided solution. As stated before CTL was used as a basis due to its feature richness. Porting CTL from x86 architecture to CBEA consisted on the first step towards the development of a STM framework for CBEA making it possible to execute transactions in the PPE processor. Here we propose the following steps in order to correctly achieve this goal.

1. The first step was to implement the communication layer between PPE and SPE, and ensure the correct usage of the memory locations allocated previously by the PPE for exchanging data with SPE.

2. The second step consisted in implementing a simple support for transactions, including managing the read/write-sets, making the integrity checks and completing the transactions successfully considering no further improvements. As a first approach we considered that the LS was able to handle all data in its 256KB, therefore the functionality tests at this point were very simple and handled small amount of data. Note that between this step and the next one a big change occurred, since in this step the SPE is able to verify for locally the write-set locally whether in the next step the write-set has to be verified remotely.

3. After the basic functionality was implemented and tested the multi-buffering techniques were implemented in order to allow to release some memory in SPE's LS. In this step the verification of the remote write-set by the SPE was implemented. The remote search is necessary since part of the write-set is remotely stored in main memory and the algorithm

requires checks on write-set to see if it the transaction has made any previous write on variable when `TxLoad` is made.

4.2 Execution Flow

In order to produce a program to use CBEA with SPE support we must compile our SPE code using a compiler that supports SPE, in this specific case `spe-gcc` GNU compiler [IBM07a], linking this compiled object into our PPE code creating the final executable.

Figure 4.1 shows how to use CTL libraries to compile the final executable that will be ran by the PPE. The programmer starts by using the SPE STM library which will produce an embedded object represented in the figure as *spu_embedded.o*. The programmer must then code the main application for the PPE using the PPE STM library and the given embedded code will produce the final executable.

Note that the embedded spu code is used by the PPE on runtime which is declared as a variable in PPE code.

Listing 4.1: Declaration of SPE embeddeded code

```
extern spe_program_handle_t spuCode;
```

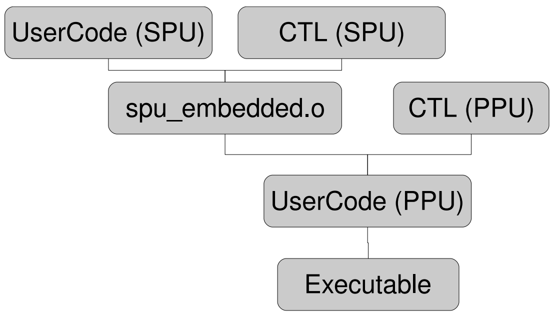

Figure 4.1: How to compile an executable.

4.3 Porting CTL from x86 to CELL

CTL [Cun07] was ported originally from a SPARC architecture to x86 on the scope of a previous MsC thesis developed by Gonçalo Cunha and it ran under the Linux OS. Linux OS also supports PowerPC architecture and CBE architecture, therefore implying minor modifications to the CTL source code in order to support the PowerPC processor present in CBEA. Taking this premises the framework was almost compatible with PowerPC except for the architecture dependent code. Since the previous implementation of CTL supported both 32 bits and 64 bits x86 processors the changes needed to support PowerPC were in the basis of architecture dependent instructions.

CTL, as a C-based STM framework, has the necessity of making atomic operations on critical sections, to guarantee integrity of data and correct execution of code. CTL uses a global clock versioning as a mechanism to maintain integrity and changes on that clock have to be atomic. This is made through gcc inline asm operations, in order to keep it atomic, more specifically it is a CAS (Compare and Swap) operation, that compares two clock timestamp values, and swaps the clock value in case a premise is true. This gcc inline asm operations are architecture dependent (or processor dependent since we are manipulating directly processor registers) and therefore had to be implemented to support PowerPC processor.

Memory Barriers were removed since we are dealing with a in-order execution [Mck05] processor, therefore eliminating the need for memory barriers, which were used the other architectures (SPARC and x86)

Some minor adjustments were made also to allow the detection of a PowerPC processor, since the objective was to maintain the compatibility with other architectures.

4.4 Extending CTL to SPE's

As stated in the solution layout (see Section 3.3), the porting of CTL to the CBEA uses the SPE's as accelerators. Using SPE's to execute computing demanding tasks as transactions in SPE environment and returning the library space data-set to the PPE in order to be validated. In order to achieve this is necessary to previously malloc the necessary library memory space in main-memory for further transfer of the transaction log. This could be made dynamically on runtime but it would increase the costs of computation since a message would had to be passed to PPE to ask for a memory location.

So in order for a transaction to execute, the library space memory is reserved for that transaction. Each call to the TxStartSPE (see Section 3.3.3) starts a thread on the PPE side which will be responsible for launching the SPU user-code into the SPE and executing it. This PPE thread launches a transaction in PPE environment, which already allocates the necessary memory for execution. The SPE will use this transaction memory space and the PPE will, on end of execution of the SPE, *Commit* the changes. The memory addresses are passed as arguments to the SPE on beginning of execution so it knows where to fetch data from/to.

Since the user needs to pass its own data-set memory addresses to his SPE execution, a structure composed by pointers to *user-space memory* and *library-space memory* is passed to the SPE which is responsibility of the STM framework. This should be interpreted by the SPE even before a `TxStart` is issued on the user-code in SPE environment since the user should be allowed to access its own memory-space pointer in non-transactional mode before any call to our STM framework. In tlCell it must be made a call to `TxStart()` in order to obtain the *user-space memory* pointer.

In order to execute transactions in the SPE's we must keep a log structure locally in the LS. They represent the read/write-set of the transaction. These are replicated as the CTL transaction structure working in PPE which will be transferred to main memory on completion. This is a sensitive issue since working in the PPE has few memory limitations contrary to working in SPE environment. Some of these problems were stated before (see Section 3.2).

The objective here is to save as much space as possible in LS for the user data-set, and executing efficiently transactions in the SPE so even though we intend to replicate the logs, not everything is necessary to be transferred to LS. Also the DMA transferences have must be multiple of 16B, or a power of two if less than 16B, so the structures must be padded in order to work.

So a log is used for the write-set and another for the read-set. We keep some of the transaction information locally in the LS like the *Thread ID* and the global clock stamp. This is necessary to the correct execution of the transaction in SPE environment. We need the global clock to abort a transaction in case of a dirty read and the thread id to assign the owner of the current transaction change into every write and read. All changes are recorded locally and then moved into main memory.

The usage of a bloom filter can be of great use [Cou09], specially in SPE environment but also has performance issues. The fact that it is possible to exclude the existence of a given data-set, both in local store and main memory reduces access time, specially regarding main memory in the Multi-Buffered model.

Lets take a deeper look at all the primitives and how they are effectively implemented in SPE context.

Note the SPE composed structure in listing 4.2 with the user struct pointer and remote library struct pointer.

Listing 4.2: Composed Structure

```
typedef struct{
    unsigned long long userStruct; //user memory space
    unsigned long long txStruct;   //library memory space
    vwLock rv;                //Versioned writeLock value.
} controlStruct_t __attribute__((aligned(128)));
```

Below in listing 4.3 are the local variables malloced in SPE for computation and the controlStruct which contains all the remote pointers to main memory. Note the instruction `SPE_ALIGN_FULL` which aligns the structures to 128 bits.

Listing 4.3: SPE control variable and local Transactional log variable

```
volatile controlStruct_t theStruct SPE_ALIGN_FULL;
volatile Tx *txStruct SPE_ALIGN_FULL;
```

- **TxStart**

Listing 4.4: TxStart pseudo-code

```
unsigned long long TxStart(long long int argp){
 mfc_get(&theStruct, argp, sizeof(theStruct));
 *txStruct =malloc(sizeof(txStruct));
 return theStruct->userStruct; //Returns user remote pointer
}
```

`TxStart()` is pretty simple. It retrieves from main memory the controlStruct (theStruct) in order to obtain the transactional remote address (theStruct->txStruct) and user remote address (theStruct->userStruct) and allocate the local structure memory space for further computation. It returns on the end the userStruct pointer.

- **TxLoad**

Listing 4.5: TxLoad pseudo-code

```
1  void TxLoad(void *destination,void *address,int size){
2   if(BloomFilterCheck(address)){
3    if(SearchWrSet(address)) return addressValue;
4   }
5
6   if(remoteAdress.readVersion<=txStruct.readVersion){
7    getRemoteAddressValue(address);
8    addEntryRdSet(address);
9    return;
10
11  else{
12   TxAbort(); //inconsistent read.
13  }
14 }
```

TxLoad receives as arguments the destination of the DMA transfer (destination), the remote address of data to retrieve (address) and the size of the transfer. In line 2 and 3 of the listing 4.5 it is checked if the correspondent remote address is already present in bloom-filter . Since bloom filters accuse false-positives we need to double check on the write-set if it is effectively present. If it is found on the write-set the correspondent address then it is returned the previously written value.

Otherwise we need to transfer from main memory the correspondent memory address. For that we need to check if we are still running in a consistent state. This is done by double checking if the versioning of the remote address is previous to our transaction timestamp. In the case where this is true (line 6 of listing 4.5) then we are running in consistent state and we can transfer the data from main-memory to local memory, proceeding with the local procedure to store the value into read-set and returning successfully.

If it is found that the remote address versioning is bigger then our transaction timestamp it means that some later transaction than ours has modified the remote address value and it is not possible to proceed with computation since we are now in an inconsistent state. Then we shall abort the transaction as stated in line 12 in listing 4.5.

- **TxStore**

Listing 4.6: TxStore pseudo-code

```
void TxStore(unsigned long long address, void *value){
 addToBloomFilter(address);
 storeEntryInWriteSet(address);
 return;
}
```

A TxStore receives as arguments the remote address `(address)` where to store the value present in the argument `value`. The address is added to the bloom filter and the operation is logged in the local transaction write-set and simply returns.

- **TxCommit(void)**

Listing 4.7: TxCommit pseudo-code

```
void TxCommit(void){
 transferReadSet();
 transferWriteSet();
 PPEValidation();
 return;
}
```

`TxCommit(void)` transfers the (remaining) read-set and write-set into main-memory so PPE is able to validate it, afterwards the PPE must validate the write/read-set in order to the transaction to commit or not. All this is done in PPE side, since the actual memory changes are made by the PPE, therefore the transaction ends and in the case where the transaction fails (it has failed write/read-set validation) then the PPE launches the transaction again.

5 Validation

5.1 Introduction

This section reports the functional validation and benchmarking results of the experimental study developed in CBEA. This section aims at studying the impact of the design choices on performance and evaluating the correctness and stability of the STM framework for CBEA.

Benchmarking a STM framework is not trivial. Usually micro-benchmarks are used to compare different STM frameworks. Although it gives a way to compare the performance between distinct frameworks usually they do not represent a real workload, complex data structures or the unpredictable behavior of a "normal" application. Fortunately there have been a serious effort to develop tools that are able to effectively benchmark STM frameworks [CMCKO08, GKV07, AKW+08], considering all the aspects like measuring commit and abort rates and runtime overheads on the applications. This tools are all complex and would require porting them for CBEA. Instead we use a series of custom made tests to ensure the Transactional Memory properties in *tlCell* framework.

Section 5.2 describes the tests made to evaluate the correctness of the STM execution. Section 5.3 refers to the performance evaluation of the experimental framework developed for CBEA.

5.2 Functional Validation

Functional Validation guarantees the correct execution of the STM framework. Any STM implementation must ensure the properties of Transactions, Atomicity, Consistency and Isolation. In order to guarantee these properties several tests were made. This includes integrity of data scoped by a transaction, persistent data across transactions, and correct reading of variables in transactional scope. In this section we present the tests made, their objectives and the results.

In order to test the main properties of a Memory Transaction — *Atomicity*, *Consistency* and *Isolation* — we prepared specific execution patterns that allows us to verify them. Lets analyze the main properties of STM and describe how does *tlCell* support each of them.

5.2.1 Atomicity

In order to assure the *Atomicity* property, a test was prepared where a set of operations is executed and one of the operations is ensured to be in conflict with another running transaction. Consequently the whole set of operations must fail and no change can be effectively made. Below we can see an example.

Listing 5.1: Transaction 1 — Conflict

```
1  int vector[nPositions];
2  int temp;
3  TxStart();
4  for(i=0;i<nPositions;i++){
5    TxLoad(temp, vector[i]);
6    TxStore(vector[i], temp++);
7  }
8
9
10 TxCommit();
```

Listing 5.2: Transaction 2 — Conflict

```
1  int vector[nPositions];
2  int temp;
3
4  TxStart();
5  TxLoad(&temp, vector[0]);
6  TxStore(vector[0], temp++);
7  TxCommit();
8
9
10 .
```

Here Transaction 1 must abort since the `TxLoad(vector[0])` is now invalid since Transaction 2 committed changes (line number 6) and Transaction 1 issued a dirty read on that variable.

It was also tested the abort scenario when changes are made but a TxAbort() is issued. Pseudo-code below in listing 5.3.

Listing 5.3: Transaction 1 - Abort

```
1  int remoteVar,temp=0;
2  TxStart();
3  TxLoad(&temp, remoteVar);
4  TxStore(remoteVar, temp++);
5  TxAbort();
6  if(remoteVar!=0) return ERR;
```

The changes made in the scope of the Transaction cannot be made visible to the rest of the system after a `TxAbort()` is issued. Therefore the variable remoteVar must stay unchanged, as stated in listing 5.3.

5.2.2 Consistency

This is a discussable point. As stated in Section 2.2.1, if we assure the *Atomicity* and *Isolation* properties we assure the *Consistency* property. Therefore no tests were directly made related to this property.

5.2.3 Isolation

This matter is of the upmost importance for any STM framework. A isolated transaction maintains a coherent view of data through the whole system. In the specific case of *tlCell*, since we are working in deferred update mode the changes are only made visible to the system after commit, which means that no other transaction will see unstable data by any other transaction. It is guaranteed that no transaction will read a *dirty value*, since it is a STM framework working in deferred update, and locks of variables are held while the read-set is validated. This invalidates any possibility of other transaction reading a value while another is updating variables. Also, *non-repeatable reads* do not happen in our case, since any read on a variable, will either check the timestamp lock, which will detect an inconsistency if any update was made in the meanwhile, or it will work on the last read value in the SPE cases. Regarding the *Phantom Reads* it is also guaranteed that no transaction will ever read sequentially two different reads, since that will also be detected on the second read (the timestamp is checked by local transaction, if any change has been made to the transaction must abort) and will invalidate the transaction.

Some pseudo-code examples that are able to detect the isolation properties are shown below. In the example 5.4 a TxLoad is issued, afterwards a TxStore on the same variable is made. The next read must reflect this change locally.

Listing 5.4: Transaction 1 — Isolation

```
int remoteVar,var;
TxStart();
TxLoad(&var,remoteVar);
TxStore(remoteVar, var++);
if(TxLoad(&var,remoteVar)!=var) return ERR;
TxCommit();
```

In the example of Figure 5.5, we check for persistent values across transactions after a successful commit.

Listing 5.5: Transaction 1 — Persistent Values

```
int remoteVar,var, var2;
TxStart();
TxLoad(&var,remoteVar);
TxStore(remoteVar, var++);
if(TxLoad(&var,remoteVar)!=var) return ERR;
TxCommit();

TxStart();
TxLoad(&var2, remoteVar);
if (var2!=var) return ERR;
TxCommit();
```

With this set of tests we ensure the ACI properties in tlCell framework.

5.2.4 Implementation Validation

While the previous tests aimed at ensuring the Transactional properties, other tests were made to ensure correct execution of the framework in very specific cases. This tests are directly related to the implementation since their objective is to guarantee a stable framework in all possible scenarios of execution. Among the several tests made different patterns were used:

- Very Short Transactions — This pattern of execution simply launches a big number of threads that make for instance only one operation over a vector.
- High Frequency of Variables Being Added and Deleted — This pattern of execution has few transactions with high number of operations of reads and writes.
- High Number of Updates on a Small Number of Variables (High contention) — This pattern execution operates on a small data set with a elevated number of threads manipulating the data set.
- More Concurrent Transactions than CPUs — This pattern executes a much higher amount of threads than the benefit that it brings, but it tests the correct interlace of transactions.

Also other tests were made regarding protection against buffer overflows (regarding the transactional logs kept in buffers for DMA transference) and bloom filter functionality and correct interlace of PPE and SPE transactions. The bloom filter was tested in order to ensure its correct functionality, remember that the bloom-filter allows us to reduce the number of searches of write-set entries, both in LS and specially in main memory which higher access time.

The interlace between PPE and SPE transactions was also tested which traduces in the possibility of using all the 9 processors (1 PPE + 8 SPEs) simultaneously taking advantage of tlCell. Although it is possible to use the PPE to execute transactions in tlCell it is used more as a controller of the framework and therefore the focus on the tests, both Functional and Performance, were on using the computational power of the SPEs.

This batch of tests was made using several distinct datatypes and the most throughly examined were the ones using a scalable vector with varying number of threads. Several operations were made in the vector, committing them on the end and issuing verifications on the data set.

With this group of tests we confirm the maintenance of Transactional Memory properties (ACI) and the correct implementation of the architectural solution.

5.3 Performance Evaluation

Once functional validation has concluded, we step into benchmarking the performance of the STM framework.

The tests were based in Ennals harness implementation([Enn05, DSS06]) which are highly configurable in terms of number of running threads, workloads (percentage of reads, writes and deletes), size of structures and time of execution. This implementation consist in either a Red-Black Tree (RBT) or a Sorted List (SL) implementation and consist of a whole of operations on a set (either RBT or SL) with variable size. It was necessary to port the previous implementation in order to support the CBEA architecture, specifically the support for Transactions in SPE context.

The test harness is divided into two components, the set implementation and the harness launcher. The set implementation refers to the RBT or SL while the harness launcher controls all the variables that are intended to be tested like execution time, number of threads and etc through arguments to the program. The variables that are possible to manipulate are shown in the Table 5.1.

The harness launcher starts a number of threads in parallel. Each thread continuously loops between: i) choosing an operation to execute (insert, remove or lookup) based on a random number; ii) calculating a random key; and iii) executing the selected operation on that key.

Table 5.1: Harness Test Configuration

Variable	**Description and limits**
Number of Threads	Number of concurrent threads
Percentage of Put,Get,Delete	Percentage of insertions, removes and gets[0 - 100]
Key range	Size of the set
Time	Time of execution

With this set of variables it is possible to measure on a specific set of data, with a specific size, and specific percentage of *inserts*, *removes* and *lookups* the throughput of the framework. The results of such test are number of successful commits and number of aborts for instance.

The patterns variations here vary between the number of executing threads, size of the working set and the percentage of inserts/removes/lookups on the set. The variation on the size of the working set is very important since it represents different contention levels. While a very large working set is likely to have collisions, a very small working set is more likely to ensure collisions. The variation of the percentage of insertions/removes/loads also aim to study the impact of different workloads on the framework.

5.3.1 PowerPC vs x86 Processor

In this section we aim to state a performance comparison between x86 and PowerPC processors only, not taking in consideration the SPEs processing capacity. The tests on x86 processors were made under the SunFire which consists on eight Dual Core processors, and the CELL tests were made on QS21b blade server using two blades, which results into two PowerPC processors with capacity of executing four concurrent threads.

The configuration used varied from the percentage of insertions, removes and lookups and number of threads using a Red-Black tree. The fixed values are the execution time of each thread, polling randomly between inserting, removing and looking up a key on a set of data. Also the size of the set was set to 1000 elements. In our result analysis we can see the three different configurations used. We measure the Throughput, which is the total of operations done (inserts, removes and lookups) and the number of aborts. The following Sections shows the results obtained.

5% Inserts - 5% Removes- 90% Lookups

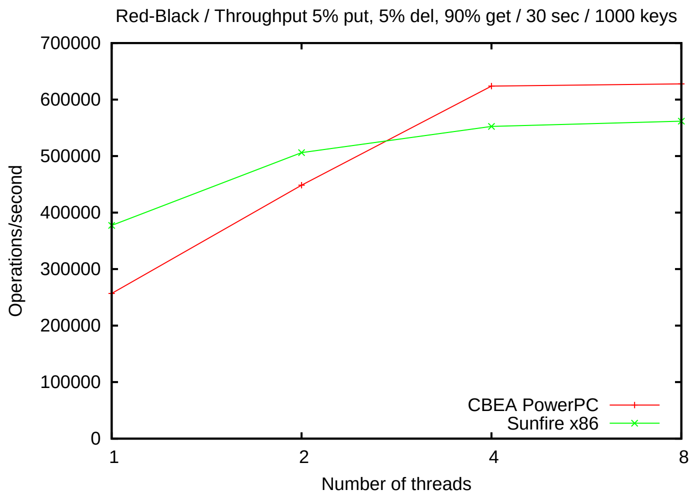

Figure 5.1: Throughput Red-Black-Tree Harness

Regarding the throughput shown in figure 5.1 we can see how PowerPC performance is linear from one to four threads, after which stops scaling, being consistent with the fact we are making the tests in a Cell Blade with two PowerPCs available which results into four possible concurrent threads. Not so consistent is the behavior of the x86 tests, which scales much less than the PowerPC, even though it has eight dual cores, resulting into sixteen possible concurrent threads. The PowerPC doubles its throughput when we double the number of threads, the same does not happen with x86.

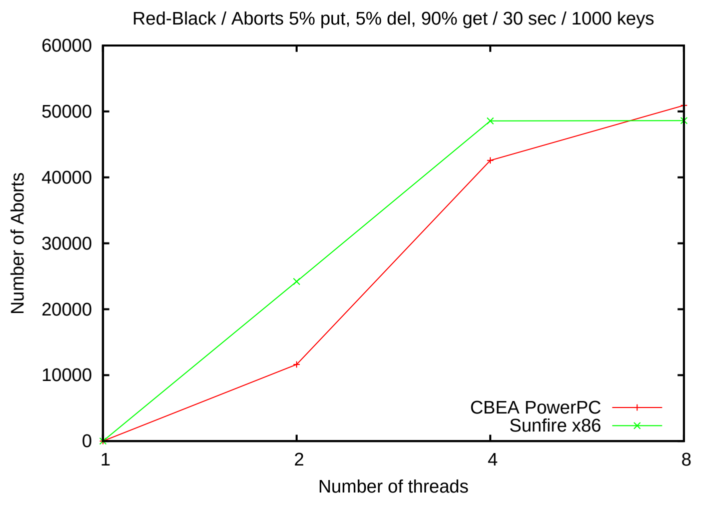

Figure 5.2: Aborts Red-Black-Tree Harness

Regarding the abort test shown in figure 5.2 both architectures behave similarly, the x86 having a linear scalability until four threads where after that drops significantly and the PowerPC quadrupling the number of aborts from two threads to four threads executions and still slightly increasing for the eight threads execution.

45% Inserts - 45% Removes- 10% Lookups

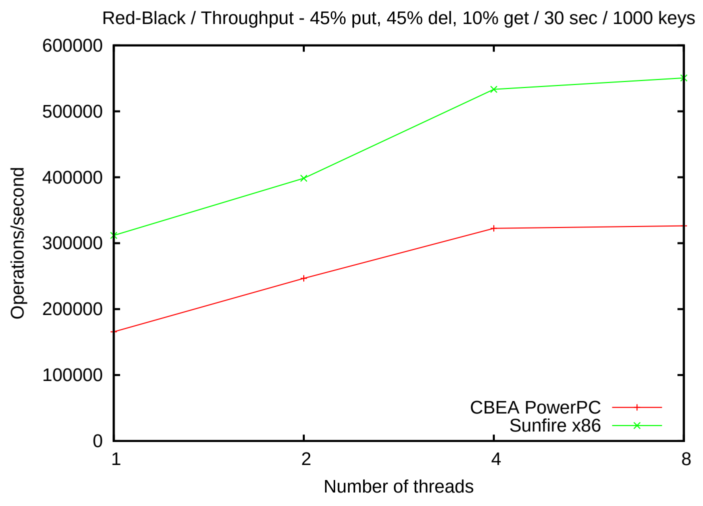

Figure 5.3: Throughput Red-Black-Tree Harness

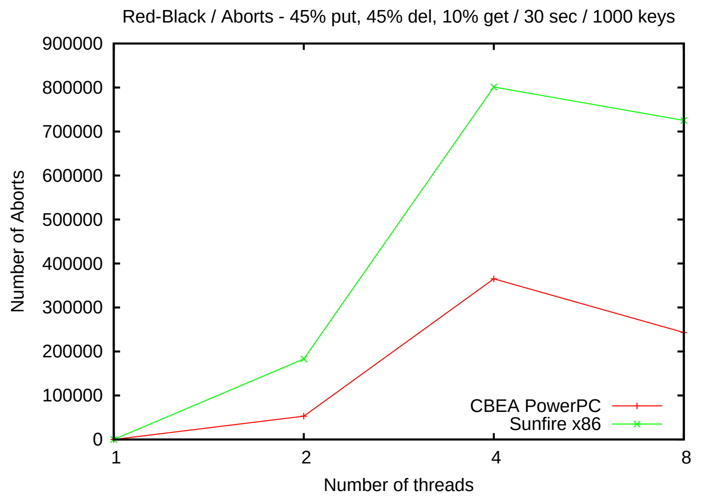

Figure 5.4: Aborts Red-Black-Tree Harness

In this configuration the x86 has a better throughput than the PowerPC as is shown in figure 5.3. Still both the architectures scale only a little until the four threads execution. After that they stabilize. Once again, as this is normal for the PowerPC, the x86 should scale after four threads, or at least more than it does.

Regarding the aborts shown in figure 5.4 both x86 and PowerPC show similar behavior, both of them approximately quadrupled the number of aborts between two threads execution and four threads execution reducing after that on the eight threads.

20% Inserts - 20% Removes- 60% Lookups

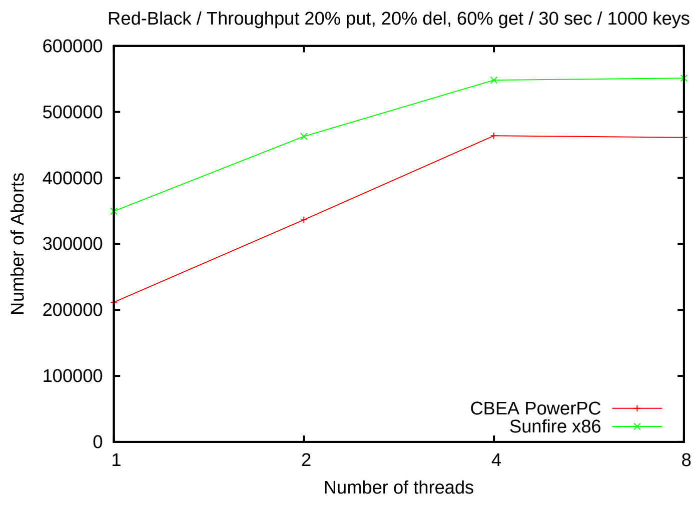

Figure 5.5: Throughput Red-Black-Tree Harness

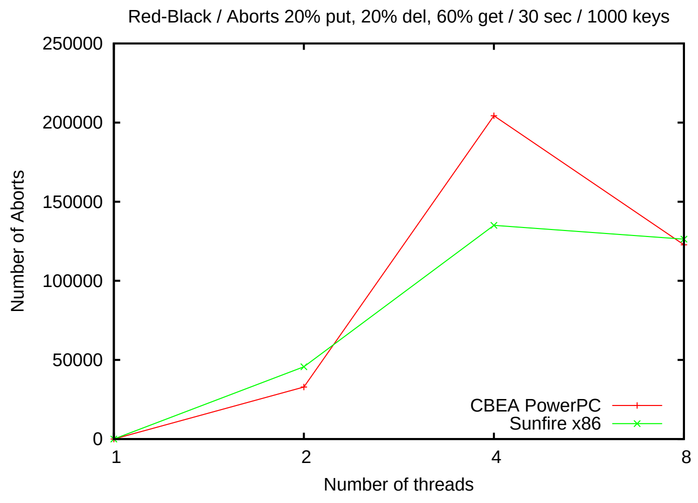

Figure 5.6: Aborts Red-Black-Tree Harness

The throughput results shown in figure 5.5 show very similar scalability between architecture, almost linear until four threads and reducing a little for the eight thread execution being the x86 the processor with better throughput.

In figure 5.6 we can see the number of aborts of both architectures. The PowerPC has a big increase between the two and four threads execution (4x more aborts) while x86 triples the number of aborts after which both decrease. The PowerPC shows a more irregular pattern, specially taking in consideration the throughput.

Remarks

Regarding the PowerPC throughput results we can verify the non-scalability between 4 and 8 threads since the tests were made with two PowerPCs which allow up to 4 threads concurrently so this results are expected. Also the number of aborts follow the same pattern, generally increasing along with the throughput.

Regarding the results on x86 platform they are less expected. The Sunfire x86 blade where the framework was tested is capable of running up to 16 threads simultaneous but that is not represented in the results. The same framework base we used for CELL was previously tested in exactly the same blade center [CLD08] under SunOS instead of Linux, achieving the expected scalability which is not shown here. In order to eliminate the possibility of changes in the framework being the cause of the problem, the base framework was tested also in Sunfire blade achieving the same results which leads to the probable conclusion of being the Linux SO. At first we though it could be related to GNU malloc instruction using locks, but we experimented other solutions not achieving any improvement.

5.3.2 Ennals Harness test in SPE

The inital objective was to port the Ennals harness test in order to measure the effeciency of transactions in SPE scope. Unfortunatelly several obstacles appeared which made the port to the SPE's harder than at first expected.

As described before the Ennals Harness test consists in the Harness section, which launches the parallel threads and measures all the information related to the execution such as total execution time, number of inserts, removes and gets, the total throughput of the execution and number of aborts. This parallel threads launched by the harness test pool for a specified amount of time executing in each cycle a random operation based on the percentage set by the user. Each of this cycles starts a transaction, an insert, get or remove commiting afterwards.

The first approach towards porting the Harness tests to the SPE's was to implement this parallel thread execution one in each SPE, polling and executing random operations. The set was prepared, aligned in memory in order to be transferable between PPEs and SPEs, the stats structure was also modified in order to allow the SPE to issue the calculations. The fact here is that tlCell only allows one Transaction execution during the whole execution in the SPE due to a design choice, since the objective was to reduce the communication between SPE and PPE as much as possible. Allowing more than one Transaction in the scope of the SPE would imply dynamic allocation of remote variables in main memory, which would increasy latency. This memory allocation mechanism will be discussed further on this section. Therefore the decision was that only one Transaction would be allowed in each SPE execution scope, meaning that the parallel thread would pool and execute the inserts, removes and gets all in one Transaction scope.

The Harness Cell test being ported was an ordered Double Linked List implementation, which means that the number of conflicts between Transactions is elevated since to find a key in the List we must start from beggining until we find it making the approach of polling in each SPE several gets, removes and inserts even more conflictuous with other Threads. This approach would traduce in nearly Serializable execution, since practically only one of the Threads would be allowed to commit at the time. Also, the insert command would have the need to allocate memory in main memory, since the set is located in main memory. Memory allocation in main memory is made through special memory mapped intruction. The instruction is the `malloc_ea` and has a high cost, so as references to any effective addres [IBM08b] so this approach turned at each point more undesirable.

So at this point the approach turned to implementing the Harness Test more like the standard launching an amount of threads in PPE context which would be launching SPE threads, each one executing at a cycle a transactional operation in the SPE, such as one get, remove or insert. This approach although it makes more sense has also a big disavantage since the cost of starting an Thread in SPE context is considerably higher than in PPE due to all the mechanisms that are necessary to initiate the process. The recomendations are to use the same thread in SPE context as much as possible in order to minimize the overhead of launching a thread, but in this context the amount of threads launched plus the rate of collisions, since the set is ordered Double Linked List, make this implementation also a bad choice. The collisions rates between the threads and the re-execution of the SPEs, which is made from PPE side would traduce also in an almost serializable execution of the harness test.

It would've been interesting to measure the results in order to really measure the cost. Unfortunatelly it was not possible in time to present here the results. Also considering all the factors stated, in order to better measure the performance possibility in SPEs it is advisable to test in a set like a Red Black Tree or a more sparse access pattern in order to avoid so many collisions since the overhead of launching threads plus the high collision rates decreases the possibility of obtaining a high performance Software Transactional Memory framework for Cell Broadband Engine Architecture.

6

Conclusions and Future Work

The development of this prototype states the possibility of using Software Transactional Memory in Cell Broadband Engine. In such complex architecture the possible mechanisms to use, improve or developing a new STM framework are numerous.

Cell Broadband Engine presents a novel architecture, unique memory distribution with a heterogeneous multiprocessor architecture and several communications mechanisms makes it very versatile and appealing for distinct Computer Science areas. The SPEs present the real potential in this architecture being used nowadays for Digital Signal Processing and High Definition rendering for example and it is mainly used in such ”problems” which take real advantage of the SIMD SPEs capacity. In general these are problems that can be data parallelized and not task parallelized. The SPEs perform poorly with scalar operations, having the need to move the data to the preffered scalar slot to perform this kind of operation, so any problem state has to take in consideration this factor when developing an application for CBEA. Using the SIMD capacity of the SPEs plus the special dedicated Memory Controllers in each SPE is the secret towards taking advantage of this architecture.

There are still several possible improvement points on this framework. The usage of a software managed cache to manage the transactional logs was a point we hoped to achieve but unfortunately was not possible and is one of the possible improvements of this prototype. Also using an Hashtable for keeping the logs could be of great advantage since it could be possible to directly access main memory to retrieve the logs, calculating the address for that data-set, instead of searching them through the PPE in the specific case of our Multi-Buffered model which decreases performance.

Besides the possible improvement points on *tlCell*, for future work a development of a STM framework for CBEA could go towards Distributed Software Transactional Memory using Two-Phase Commit protocol for instance. Studying the possibility of implementing a direct-update STM framework could be interesting also due to the hardware mechanisms that CBEA provide for Atomic Operations through DMA operations.

In this dissertation we hope to contribute to the scientific community by introducing *tlCell* and stating all the problems we have encountered during the development of the prototype, evaluating the pros and cons of the framework design choices.

Bibliography

[AKW$^+$08] Mohammad Ansari, Christos Kotselidis, Ian Watson, Chris Kirkham, Mikel Luján, and Kim Jarvis. Lee-tm: A non-trivial benchmark suite for transactional memory. In *ICA3PP '08: Proceedings of the 8th international conference on Algorithms and Architectures for Parallel Processing*, pages 196–207, Berlin, Heidelberg, 2008. Springer-Verlag.

[BLKD07] Alfredo Buttari, Piotr Luszczek, Jakub Kurzak, and Jack Dongarra. Scop3: A rough guide to scientific computing on the playstation 3. Technical report, Innovative Computing Laboratory, University of Tennessee Knoxville, 2007.

[BLM05] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *In Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.

[CLD08] G. Cunha, J. Lourenço, and R. Dias. Consistent state software transactional memory. In *Proceedings of IV Jornadas de Engenharia de Electrónica e Telecomunicações e de Computadores (JEETC'2008)*. Instituto Superior Politécnico de Lisboa, November 2008.

[CMCKO08] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[Cou09] Maria Couceiro. Cache coherence in distributed and replicated transactional memory systems. Technical report, Instituto Superior Tecnico, 2009.

[Cun07] Goncalo Cunha. Consistent state software transactional memory. Technical report, FCT-UNL, Lisbon, Portugal, 2007.

[DSS06] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *In Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[Enn05] R Ennals. Software transactional memory should not be obstruction-free. In ., 2005.

[GKV07] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315–324, 2007.

[Han77] Per Brinch Hansen. *The architecture of concurrent programs*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1977.

[HEM93] Maurice Herlihy, J. Eliot, and B. Moss. Transactional memory: architectural support for lock-free data structures. In *in Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.

[IBM07a] IBM. Cell broadband engine architecture. Technical report, 2007.

[IBM07b] IBM. Cell broadband engine programming handbook. Technical report, IBM, 2007.

[IBM07c] IBM. Cell broadband engine registers. Technical report, IBM, 2007.

[IBM07d] IBM. Software development kit for multicore acceleration version 3.0 programming tutorial. Technical report, IBM, 2007.

[IBM08a] IBM. C/c++ language extensions for cell broadband engine architecture. Technical report, IBM, 2008.

[IBM08b] IBM. Spu runtime library extensions - programmer's guide and api reference. Technical report, IBM, 2008.

[JGMR07] D. Jimenez-Gonzalez, X. Martorell, and A. Ramirez. Performance analysis of cell broadband engine for high memory bandwidth applications. *Performance Analysis of Systems and Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 210–219, April 2007.

[Lom77] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *SIGPLAN Not.*, 12(3):128–137, 1977.

[LR06] James R. Larus and Ravi Rajwar. Transactional memory. *Synthesis Lectures on Computer Architecture*, 1(1):1–226, 2006.

[Mck05] Paul E. Mckenney. Memory ordering in modern microprocessors. *Linux Journal*, 30:52–57, 2005.

[MMA06] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 198–208, New York, NY, USA, 2006. ACM.

[Mos81] E. B. Moss. Nested transactions: An approach to reliable distributed computing. Technical report, ., Cambridge, MA, USA, 1981.

[Noe] Cyprien Noel. Extended software transactional memory. Technical report.

[Pet05] Hofstee H. Peter. Power efficient processor architecture and the cell processor. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262, Washington, DC, USA, 2005. IEEE Computer Society.

[Sco06] M. L. Scott. Sequential specification of transactional memory semantics. In *In TRANSACT:First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[SKS05] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Publishing Co., August 2005.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[WSMF03] Joe Wetzel, Ed Silha, Cathy May, and Brad Frey. Powerpc user instruction set architecture. Technical report, IBM, 2003.