

Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG-3)

January 24, 2010 Pisa, Italy

**In Conjunction with the Fifth International
Conference on High-Performance Embedded
Architectures and Compilers (HIPEAC)**

Message from the Organizers

Welcome to the Third Workshop on Programmability Issues for Multi-Core Computers.

We are delighted to present a strong program composed of 7 high-quality papers and a great Keynote talk on Transactional Memory by Tim Harris of Microsoft Research Cambridge.

As computer manufacturers are embarking on the multi-core roadmap, which promises a doubling of the number of processors on a chip every other year, the programming community is faced with a severe dilemma. Until now, software has been developed with a single processor in mind and it needs to be parallelized to take advantage of the new breed of multi-core computers. As a result, progress in how to easily harness the computing power of multi-core architectures is in great demand.

This workshop brings together researchers interested in programming models and their implementation and in computer architecture; who share a common interest in advancing our knowledge on how to simplify the task of parallelization of software for multi-core platforms. A wide spectrum of issues are central themes for this workshop; such as what the future programming models should look like to accelerate software productivity and how it should be implemented at the runtime, the compiler, and the architecture level.

We would like to thank the Program Committee and the additional reviewers for their hard work, almost all submissions had three reviews each.

**Eduard Ayguadé (BSC/UPC), Roberto Gioiosa (BSC),
Per Stenström (Chalmers), Osman S. Ünsal (BSC)**

Organizing Committee

Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)

Advance program

09:00-10:00 **KEYNOTE:**

Making Sense of Transactional Memory, Tim Harris, Microsoft Research Cambridge

10:00-10:30 **BEST PAPER:**

Noninvasive concurrency with Java STM, G. Korland (Tel Aviv U.) N. Shavit (Tel Aviv U.), P. Felber (U. of Neuchatel)

10:30-11:00 *Coffee break*

11:00-12:30 **SESSION 1:**

Parallelizing Barnes-Hut Method on the Cell BE Architecture, B. Demioz (Bogazici U.), H. Topcuoglu (Marmara U.), M. Kandemir (Pennsylvania State U.), O. Tosun (Bogazici U.)

Expressing Inter-task Dependencies between Parallel Stencil Operations, P. Larsen, S. Karlsson, J. Madsen (Dept. of Informatics, Technical University of Denmark)

A Performance Comparison of some recent Task-based Parallel Programming Models, A. Podobas (KTH), M. Brorsson (KTH and Swedish Institute of Computer Science), K. Faxén (Swedish Institute of Computer Science)

12:30-14:00 *Lunch*

14:00-15:30 **SESSION 2:**

Handling of shared memory in many-core processors without locks and transactional memory A. Vajda (Ericsson)

J-DSE: Joint Software and Hardware Design Space Exploration for Application Specific Processors M. Paolieri (BSC), I. Bonesana (SUPSI), R. Gioiosa (BSC), M. Valero (UPC / BSC)

Building a Java Map-Reduce Framework for Multi-core Architectures G. Kovoor; J. Singer, M. Lujan (U. of Manchester)

15:30-16:00 *Closing*

Noninvasive concurrency with Java STM

Guy Korland¹ and Nir Shavit¹ and Pascal Felber²

¹ Computer Science Department
Tel-Aviv University, Israel,

`guykorla@post.tau.ac.il`, `shanir@post.tau.ac.il`

² University of Neuchâtel, Neuchâtel, Switzerland,
`pascal.felber@unine.ch`

Abstract. In this paper we present a complete Java STM framework, called DEUCE, intended as a platform for developing scalable concurrent applications and as a research tool for designing new STM algorithms. It was not clear if one could build an efficient Java STM without compiler support. DEUCE provides several benefits over existing Java STM frameworks: it avoids any changes or additions to the JVM, it does not require language extensions or intrusive APIs, and it does not impose any memory footprint or GC overhead. To support legacy libraries, DEUCE dynamically instruments classes at load time and uses an original “field-based” locking strategy to improve concurrency. DEUCE also provides a simple internal API allowing different STMs algorithms to be plugged in. We show empirical results that highlight the scalability of our framework running benchmarks with hundreds of concurrent threads. This paper shows for the first time that one can actually design a Java STM with reasonable performance without compiler support.

1 Introduction

Multicore CPUs have become commonplace, with dual-cores powering almost any modern portable or desktop computer, and quad-cores being the norm for new servers. While multicore hardware has been advancing at an accelerated pace, software for multicore platforms seems to be at a crossroads.

Currently, two diverging programming methods are commonly used. The first exploits concurrency by synchronizing multiple threads based on locks. This approach is well known to be a two-edged sword: on the one hand, locks give the programmer a fine-grained way to control the applications critical sections, allowing an expert to provide great scalability; on the other hand, because of the risk of deadlocks, starvation, priority inversions, etc., they impose a great burden on non-expert programmers, often leading them to prefer the safer but non-scalable alternative of coarse-grained locking.

The second method is a shared-nothing model usually common in Web based architectures. The applications usually contain only the business logic deployed in a container, while the state is saved in an external multi-versioned control system, such as database, message queue, or distributed cache. While this method removes the burden of handling concurrency in the application, it imposes a huge performance impact on data accesses and is not seamlessly applicable to many types of application.

The hope is that transactional memory (TM) will simplify concurrent programming by combining the desirable features of both methods, providing *state-aware shared memory* with *simple concurrency control*.

In the past several years there has been a flurry of software transactional memory (STM) design and implementation work. The state of the art today however is less than appealing. With the notable exception of transactional C/C++ compilers [9], most of STM initiatives have remained academic experiments, applicable only to “toy applications”, and though we have learned much from the process of developing them, they never reached a state that will allow them to be seriously field tested. There are several reasons for this. Among them are the problematic handling of many features of the target language, the large performance overheads, and the lack of support for legacy libraries. Moreover, many of the published results have been based on prototypes whose source code is unavailable or poorly maintained, making a reliable comparison between the various algorithms and designs very difficult.

In this paper, we introduce DEUCE, our novel open-source Java framework for transactional memory. DEUCE has several desired features not found in earlier Java STM frameworks. As we discuss in Section 2, there currently does not exist an *efficient* Java STM framework that delivers a full feature STM that can be added to an existing application *without changes* to its compiler or libraries. It was not clear if one could build an efficient Java STM without compiler support.

DEUCE is intended to fill this gap. It is non-intrusive in the sense that no modifications to the Java virtual machine (JVM) or extensions to the language are necessary. It uses, by default, an original locking design that detects conflicts at the level of individual *fields* without a significant increase in the memory footprint (no extra data is added to any of the classes) and therefore there is no GC overhead. This locking scheme provides finer granularity and better parallelism than former object-based lock designs. In order to avoid performance penalty, DEUCE provides weak atomicity, i.e., it does not guarantee that concurrent accesses to a shared memory location from both inside and outside a transaction are consistent. It also supports a pluggable STM back-end and an open easily extendable architecture, allowing researchers to integrate and test their own STM algorithms.

DEUCE has been heavily optimized for efficiency and, while there is still room for improvements, our performance evaluations on several high-end machines (up to 128 hardware threads on a 16-core Sun Niagara-based machine and a 96-core Azul machine) demonstrate that it scales well. Our benchmarks show that it outperforms the main competing JVM-independent Java STM, the DSTM2 framework [6], in many cases by two orders of magnitude, and in general scales well on many workloads. This paper shows for the first time that one can actually design a Java STM with reasonable performance without compiler support.

The DEUCE framework has been in development for more than a year, and we believe it has reached a level of maturity sufficient to allow it to be used by developers with no prior expertise in transactional memory. It is our hope that DEUCE can help democratize the use of STM among developers, and that its open-source nature will encourage STM them to extend the infrastructure with novel features, as has been the case, for instance, with the Jikes RVM [1].

The rest of the paper is organized as follows. We first overview related work in Section 2. Section 3 describes DEUCE from the perspective of the developer of a concurrent application. In Section 4 we discuss the implementation of the framework, and we show how it can be extended by the means of the STM backends in Section 5. Finally, in Section 6, we evaluate the performance of DEUCE.

2 Related work

Several tools have been proposed to allow the introduction of STM into Java. They differ in their programming interface and in the way they are implemented. One of the first proposals is Harris and Fraser’s CCR [5] that used a C-based STM implementation underneath the JVM and a programmatic API to use STM in Java.

DSTM [7] is another pioneering Java STM implementation. It used an explicit API to demarcate transactions and read/write shared data. Transactional objects had to additionally provide some pre-defined functionality for cloning their state.

More recently, the DSTM2 [6] framework was introduced, proposing a set of higher-level mechanisms to support transactions in Java. DSTM2 is a pure Java library that does not require any change to the JVM nor to the compiler. It uses a special annotation to mark an atomic interface. Only a class that implements an annotated interface can participate in a transaction. This class must be created using a special factory and it can only support transactional access to primitive types or other annotated classes. Transactions must be started using a `Callable` object. This design creates an API that requires important refactoring of legacy code and introduces many limitations, such as the lack of support for existing libraries.

LSA-STM [12] is a Java STM that also relies primarily on annotations for TM programming. Transactional objects to be accessed in the context of a transaction must be annotated as `@Transactional` and methods must be declared as `@Atomic`. Additional annotations can be used to indicate that some methods will not modify the state of the target object (read-only) or to specify the behavior in case an exception is thrown within a transaction. Transactional objects are implemented in LSA-STM using inheritance and must support state cloning, as such it does not fully support legacy code and is not transparent nor non-invasive as DEUCE is. Among the limitations of the framework, one can mention that accesses to a field are only supported as part of methods from the owner class, therefore public and static fields cannot be accessed transactionally. Note that this limitation also applies to other Java STM implementations with object-level conflict detection like DSTM and DSTM2.

AtomJava [8] takes a different approach to Java STM design by providing a source-to-source translator based on Polyglot [10], an extensible compiler framework. AtomJava adds a new `atomic` keyword to mark atomic blocks, and performs some major extensions during code transformation, such as adding an extra instance field to every single class. AtomJava relies on this field to maintain an object-based lock schema. It is shown to reduce lock access overhead but imposes a memory overhead which impacts not only transactional objects but all the application objects. The source code translation approach simplifies the tuning and verification of STM instrumentation, but it makes it harder for the programmer to debug her original code. Also, by translating source code, AtomJava imposes a major limitation on users in that it prevents them from using compiled libraries, and renders their source code with atomic blocks incompatible with regular Java compilers (unlike approaches such as DSTM2 and LSA-STM that are based on annotations).

In summary, DEUCE is the first *efficient full-feature* STM framework that can be added to an existing application *without changing its compilation process or libraries*.

3 Concurrent Programming with DEUCE

One of the main goals in designing the DEUCE API was to keep it simple. A survey of the past designs (see previous section) reveals three main approaches: (i) adding a new reserved keyword to mark a block as atomic, e.g., `atomic` in AtomJava [8]; (ii) using explicit method calls to demarcate transactions and access shared objects, e.g., DSTM2 [6] and CCR [5]; or (iii) requiring atomic classes to implement a pre-defined interface or extend a base class, and to implement a set of methods [7]. The first approach requires modifying the compiler and/or the JVM, while the others are intrusive from the programmer’s perspective as they require significant modifications to the code (even though some systems use semi-automatic weaving mechanisms such as aspect-oriented programming to ease the task of the programmer, e.g., [12]).

In contrast, DEUCE has been designed to avoid any addition to the language or changes to the JVM. In particular, no special code is necessary to indicate which objects are transactional, no method needs to be implemented to supporting transactional objects (e.g., cloning the state as in [7]). This allows DEUCE to seamlessly support transactional accesses to compiled libraries. The only piece of information that must be specified by the programmer is, obviously, which part of the code should execute atomically in the context of a transaction.

To that end, DEUCE relies on Java annotations. Introduced as a new feature in Java 5, annotations allow programmers to mark a method with metadata that can be consulted at class loading time. DEUCE introduces new types of annotations to mark methods as atomic: their execution will take place in the context of a transaction.

This approach has several advantages, both technically and semantically. First technically, the smallest code block Java can annotate is a method, which simplifies the instrumentation process of DEUCE and provides a simple model for the programmer. Second, atomic annotations operate at the same level as *synchronized* methods, which execute in a mutually exclusion manner on a given object; therefore, atomic methods provide a familiar programming model.

From a semantic point of view, implementing atomic blocks at the granularity of methods removes the need to deal with local variables as part of the transaction. In particular, since Java doesn’t allow any access to stack variables outside the current method, the STM can avoid logging many memory accesses. For instance, in Figure 1, a finer-granularity atomic block would require costly logging of the `total` local variable (otherwise the method would yield incorrect results upon abort) whereas no logging would be necessary when considering atomic blocks at the granularity of individual methods.

To illustrate the use and implementation of DEUCE, we will consider a well-known but non-trivial data structure: the skip list [11]. A skip list is a probabilistic structure based on multiple parallel, sorted linked lists, with efficient search time in $\mathcal{O}(\log n)$.

Figure 2 shows a partial implementation of skip list, with an inner class representing nodes and a method to search for a value through the list. The key observation in this code is that transactifying an application is as easy as adding `@Atomic` annotations to methods that should execute as transactions. No code needs to be changed

```
1 public int sum(List list) {  
2   int total = 0;  
3   atomic {  
4     for (Node n : list)  
5       total += n.getValue();  
6   }  
7   return total;  
8 }
```

Fig. 1. Atomic block example.

within the method or elsewhere in the class. Interestingly, the linked list directly manipulates arrays and accesses public fields from outside their enclosing objects, which would not be possible with DSTM2 or LSA-STM.

```

1 public class SkipList {
2     private static class Node {
3         public final int value;
4         public final Node[] forward;
5         // ...
6         public Node(int level, int v) {
7             value = v;
8             forward = new Node[level + 1];
9         }
10        // ...
11    }
12    private static int MAX_LEVEL = 32;
13    private int level;
14    private final Node head;
15    // Continued in next column...
16    @Atomic(retries=64)
17    public boolean contains(int v) {
18        Node node = head;
19        for (int i = level; i >= 0; i--) {
20            Node next = node.forward[i];
21            while (next.value < v) {
22                node = next;
23                next = node.forward[i];
24            }
25        }
26        node = node.forward[0];
27        return (node.value == v);
28    }
29    // ...
30 }

```

Fig. 2. @Atomic method example.

One can also observe that the @Atomic annotation provides one configurable attribute, `retries`, to optionally limit the amount of retries the transaction attempts (at most 64 times in the example). A `TransactionException` is thrown in case this limit is reached. Alternatively one can envision providing timeout instead (Which we might add in future versions).

A DEUCE application is compiled with a regular Java compiler. Upon execution, one needs to specify a Java agent that allows DEUCE to intercept every class loaded and manipulate it before it is loaded by the JVM. The agent is simply specified on the command line as a parameter to the JVM, as:

```
java -javaagent:deuceAgent.jar MainClass args...
```

As will be discussed in the next section, DEUCE instruments every class that may be used from within a transaction, not only classes that have @Atomic annotations. If it is known that a class will never be used in the context of a transaction, one can prevent it from being instrumented by providing exclusion lists to the DEUCE agent. This will speed up the application loading time yet should not affect execution speed.

4 DEUCE Implementation

This section describes the implementation of the DEUCE framework. We first give a high-level overview of its main components. Then, we explain the process of code instrumentation. Finally, we describe various optimizations that enhance the performance of transactional code.

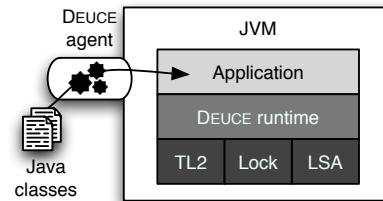


Fig. 3. Main components of the DEUCE architecture.

4.1 Overview

The DEUCE framework is conceptually made of 3 layers, as shown in Figure 3:

1. The application, which consists of user classes written without any relationship to the STM implementation, except for annotations added to atomic methods.
2. The DEUCE runtime that orchestrates the interactions between transactions executed by the application and the underlying STM implementation.

3. The actual STM libraries that implement the DEUCE context API (see Section 5), including a reference single-global-lock implementation (“Lock” in the Figure).

In addition, the DEUCE agent intercepts classes as they are loaded and instruments them before they start executing.

4.2 Instrumentation Framework

DEUCE’s instrumentation engine is based on ASM [3], an all-purpose Java bytecode manipulation and analysis framework. It can be used to modify existing classes or to dynamically generate classes, directly in binary form. The DEUCE Java agent uses ASM to dynamically instrument classes as they are loaded, before their first instance is created in the virtual machine. During instrumentation, new fields and methods are added, and the code of transactional methods is transformed. We now describe the different manipulations performed by the Java agent.

Fields. For each instance field in any loaded class, DEUCE adds a synthetic constant field (`final static public`) that represents the relative position of the field in the class. This value, together with the the instance of the class, uniquely identifies a field and is used by the STM implementation to log field accesses.

Static fields in Java are effectively fields of the enclosing class. To designate static fields, DEUCE defines for each class a constant that represents the base class and can be used in combination with the field position, instead of the class instance.

```

1 public class SkipList {
2   public static final long __CLASS_BASE__ = ...
3   public static final long MAX_LEVEL__address__ = ...
4   public static final long level__address__ = ...
5   // ...
6   private static int MAX_LEVEL = 32;
7   private int level;
8   // ...
9 }

```

Fig. 4. Fields address.

For instance, in Figure 4, the `level` field is represented by `level__ADDRESS__` while the `SkipList` base class is represented by `__CLASS_BASE__`.

Accessors. For each field of any loaded class, DEUCE adds synthetic `static` accessors used to trigger field’s access events on the local context.

Figure 5 shows the synthetic accessors of class `SkipList`. The *getter* `level__Getter$` receives the current field value and a context, and triggers two events: `beforeReadAccess` and `onReadAccess`; the result of the latter is returned by the getter. The *setter* receives a context and yields a single event: `onWriteAccess`. The reason for having two events in the getter is technical: the

```

1 public class SkipList {
2   private int level;
3   // ...
4
5   // Synthetic getter
6   public int level__Getter$(Context c) {
7     c.beforeReadAccess(this, level__ADDRESS__);
8     return c.onReadAccess(this, level, level__ADDRESS__);
9   }
10  // Synthetic setter
11  public void level__Setter$(int v, Context c) {
12    c.onWriteAccess(this, v, level__ADDRESS__);
13  }
14  // ...
15 }

```

Fig. 5. Fields accessors.

“before” and “after” events allow the STM backend to verify that the value of the field, which is accessed between both events without using costly reflection mechanisms, is consistent. This is typically the case in time-based STM algorithms like TL2 and LSA that ship with DEUCE.

Duplicate methods. In order to avoid any performance penalty on non-transactional code and since DEUCE provides weak atomicity, DEUCE duplicates each method to provide two distinct versions. The first version is identical to the original method: it does not trigger an event upon memory access and, consequently, does not impose any transactional overhead. The second version is a synthetic method with an extra `Context` parameter. In this instrumented copy of the original method, all field accesses (except for `final` fields) are replaced by calls to the associated transactional accessors. Figure 4 shows the two versions of a method of class `Node`. The second synthetic overload replaces `forward[level] = next` by calls to the synthetic getter and setter of the `forward` array. The first call obtains the reference to the array object, while the second one changes the specified array element (note that getters and setters for array elements have an extra index parameter).

```

1 private static class Node {
2     public Node[] forward;
3     // ...
4
5     // Original method
6     public void setForward(int level, Node next) {
7         forward[level] = next;
8     }
9
10    // Synthetic duplicate method
11    public void setForward(int level, Node next, Context c) {
12        Node[] f = forward__Getter$(c) {
13            forward__Setter$(f, level, c)
14        }
15    // ...
16 }

```

Fig. 6. Duplicate method.

```

1 public class SkipList {
2     // ...
3
4     // Original method instrumented
5     public boolean contains(int v) {
6         Throwable throwable = null;
7         Context context =
8             ContextDelegator.getInstance();
9         boolean commit = true;
10        boolean result;
11
12        for (int i = 64; i > 0; --i) {
13            context.init ();
14            try {
15                result = contains(v, context);
16            } catch (TransactionException ex) {
17                // Must rollback
18                commit = false;
19            } catch (Throwable ex) {
20                throwable = ex;
21            }
22            // Continued in next column...
23
24            // Try to commit
25            if (commit) {
26                if (context.commit()) {
27                    if (throwable == null)
28                        return result;
29                    // Rethrow application exception
30                    throw (IOException)throwable;
31                } else {
32                    context.rollback ();
33                    commit = true;
34                }
35            } // Retry loop
36            throw new TransactionException();
37        }
38
39        // Synthetic duplicate method
40        public boolean contains(int v, Context c) {
41            Node node = head__Getter$(c);
42            // ...
43        }
44    }

```

Fig. 7. Atomic method.

Atomic methods. The duplication process described above has one exception: a method annotated as `@Atomic` does not need an uninstrumented version. Instead, the original method is replaced by a transactional version that calls the instrumented version from within a transaction that executes in a loop. The process repeats as long as the transaction aborts and the number of retries is not reached. Figure 7 shows the transactional version of method `contains`.

4.3 Optimizations

During instrumentation, we perform several optimizations to improve the performance of DEUCE. First, we do not instrument accesses to final fields as they cannot be modified after creation. This optimization, together with the declaration of final fields whenever possible in the application code, dramatically reduces the overhead.

Second, instead of generating accessor methods, DEUCE actually inlines the code of the getters and setters directly in the transactional code. We have observed a slight performance improvement from this optimization.

Third, we chose to use the `sun.misc.Unsafe` pseudo-standard internal library to implement fast reflection, as it proved to be vastly more efficient than the standard Java reflection mechanisms. Using `sun.misc.Unsafe` even outperformed the approach taken in AtomJava [8], which is based on using an anonymous class per field to replace reflection.

Finally, we tried to limit as much as possible the stress on the garbage collector, notably by using object pools when keeping track of accessed fields (read and write sets) in threads. In order to avoid any contention on the pool, we had each thread keep a separate object pool as part of its context.

Together, the above optimizations helped to significantly decrease the implementation overhead, in some of our benchmarks by almost an order of magnitude (almost ten times faster) as compared to our initial implementation.

5 Customizing Concurrency Control

DEUCE was designed to provide a research platform for STM methods. In order to provide a simple API for researchers to plug in their own STM implementation, DEUCE defines the `Context` API as shown in Listing 8. The API includes an `init` method called once before the transaction starts and then upon each retry, allowing the transaction to initialize its internal data structures. The `atomicBlockId` argument allows the transaction to log information about the specific atomic block (statically assigned in the bytecode).

One of the heuristics we added to the LSA implementation is, following [4], that each one of the atomic blocks will initially be a read-only block. It will be converted to become a writable block upon retry once it encounters a first write. Using this method, read-only blocks can save most of the overhead of logging the fields' access.

```
1 public interface Context {
2     void init(int atomicBlockId);
3     boolean commit();
4     void rollback();
5
6     void beforeReadAccess(Object obj, long field);
7
8     Object onReadAccess(Object obj, Object value, long field);
9     int onReadAccess(Object obj, int value, long field);
10    long onReadAccess(Object obj, long value, long field);
11    // ...
12
13    void onWriteAccess(Object obj, Object value, long field);
14    void onWriteAccess(Object obj, int value, long field);
15    void onWriteAccess(Object obj, long value, long field);
16    // ...
17 }
```

Fig. 8. Context interface.

Another heuristic is that `commit` is called in case the atomic block finishes without a `TransactionException` and can return `false` in case the commit fails, which in

turn will cause a retry. A `rollback` is called when a `TransactionException` is thrown during the atomic block (this can be used by the business logic to force a retry).

The rest of the methods are called upon field access: a field read event will trigger a `beforeReadAccess` event followed by a `onReadAccess` event. A write field event will trigger an `onWriteAccess` event. DEUCE currently includes two `Context` implementations, `tl2.Context` and `lsa.Context`, implementing the TL2 [4] and LSA [12] STM algorithms. DEUCE also provides a reference implementation based on a single global lock. Since a global lock doesn't log any field access, it doesn't implement the `Context` interface and doesn't impose any overhead on fields' access.

TL2 Context. The TL2 context is a straightforward implementation of the TL2 algorithm [4]. The general principle is as follows (many details omitted).

TL2 uses a shared array of *revokable versioned locks*, with each object field being mapped to a single lock. Each lock has a version that corresponds to the commit timestamp of the transaction that last updated some field protected by this lock. Timestamps are acquired upon commit from a global time base, implemented as a simple counter.

When reading some data, a transaction checks that the associated timestamp is valid, i.e., not more recent than the time when the transaction started, and keeps track of the accessed location in its read set. Upon write, the transaction buffers the update in its write set.

At commit time, the transaction acquires (using an atomic compare-and-set operation) the locks protecting all written fields, verifies that all entries in the read set are still valid, acquires a unique commit timestamp, writes the modified fields to memory, and releases the locks. If locks cannot be acquired or validation fails, the transaction aborts, discarding buffered updates.

LSA Context. The LSA context uses the LSA algorithm [12] that was developed concurrently with TL2 and is based on a similar design. The main differences are that (1) LSA acquires locks as fields are written, instead of at commit time, and (2) it performs "incremental validation" to avoid aborting transactions that read data that has been modified more recently than the start time of the transaction.

Both TL2 and LSA take a simple approach to conflict management: they simply abort and restart (possibly after a delay) the transaction that encounters the conflict. This strategy is simple and efficient to implement, because transactions unilaterally abort without any synchronization with others, but it can sometimes hamper progress, e.g., by producing livelocks. DEUCE also supports a variant of the LSA context that provides modular contention management strategies (as first proposed in [7]), at the price of some additional synchronization overhead at runtime.

6 Performance Evaluation

We evaluated the performance of DEUCE on a Sun UltraSPARC T2 Plus multicore machine (2 CPUs each with 8 cores at 1.2 GHz, each with 8 hardware threads) and an Azul Vega2 (2 CPUs each with 48 cores).

6.1 DEUCE overheads

We first briefly discuss the overheads introduced by the DEUCE agent when instrumenting Java classes. Table 1 shows the memory footprint and the processing overhead when processing compiled libraries: `rt.jar` is the runtime library containing all Java built-in classes for JDK 6; `cache4j.jar` (version 0.4) is a widely used in-memory cache for Java objects; *JavaGrande* (version 1.0) is a well-known collection of Java benchmarks. Instrumentation times were measured on a x86 Core 2 Duo CPU running at 1.8 GHz. Note that instrumentation was executed serially, i.e., without exploiting multiple cores.

Application	Memory		Instrumentation Time
	Original	Instrumented	
<code>rt.jar</code>	50 MB	115 MB	29 s
<code>cache4j.jar</code>	248 KB	473 KB	<1 s
<i>JavaGrande</i>	360 KB	679 KB	<1 s

Table 1. Memory footprint and processing overhead.

As expected, the size of the code approximately doubles because DEUCE duplicates each method, and the processing overhead is proportional to the size of the code. However, because Java uses lazy class loading, only the classes that are actually *used* in an application will be instrumented at load time. Therefore, the overheads of DEUCE are negligible considering individual classes are loaded on demand.

In terms of execution time, instrumented classes that are not invoked within a transaction incur no performance penalty as the original method executes, not the transactional version.

6.2 Benchmarks

We tested the three built-in DEUCE STM options: LSA, TL2, and a simple Global Lock. Our LSA version captures many of the properties of the LSA-STM [13] framework. The TL2 [4] form provides an alternative STM implementation that acquires locks at commit time using a write set as opposed to the encounter order and undo log approach used in LSA.

Our experiments included three classical micro-benchmarks: a red/black tree, a sorted linked list and a skip list, on which we performed concurrent insert, delete, and search operations. We controlled the size of the data structures, which remained constant for the duration of the experiments, and the mix of operations.

We also experimented with two real-world applications, the first implements the Lee routing algorithm (as presented in Lee-TM [2]). It is worth noting that adapting the application to DEUCE was trivial, with little more than a change of synchronized blocks into atomic methods. The second application called Cache4J, is an LRU cache implementation, again adapting the application to DEUCE was trivial.

Micro-benchmarks. We began by comparing DEUCE to DSTM2, the only competing STM framework that does not require compiler support. We benchmarked DSTM2 and DEUCE based on the Red/Black tree benchmark provided with the DSTM2 framework. We tried many variations of operation combinations and levels of concurrency on both the Azul and Sun machines. Comparison results for a tree with 10k elements are shown in Figure 9, we found that in all the benchmarks DSTM2

was about 100 times (two orders of magnitude) slower than DEUCE. Our results are consistent with those published in [6], where DSTM2's throughput in operations per second is in the thousands while DEUCE throughput is in the millions. Based on these experiments, we believe one can conclude that DEUCE is the first viable compiler and JVM independent Java STM framework.

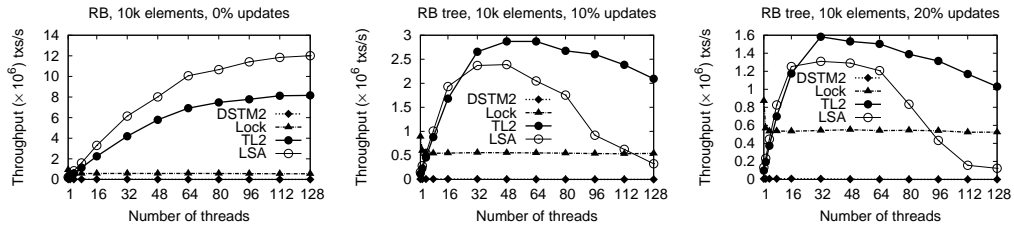


Fig. 9. The red/black tree benchmark (Sun).

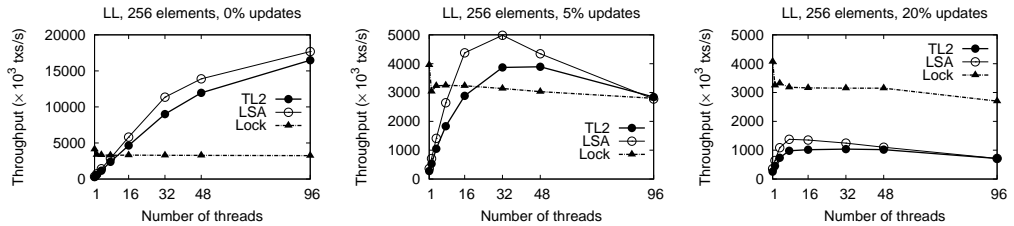


Fig. 10. The linked list benchmark (Sun).

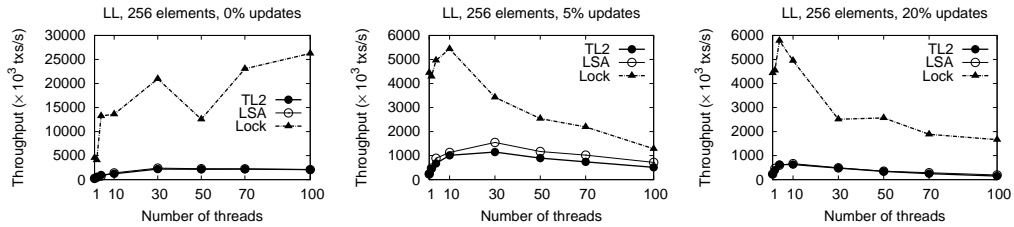


Fig. 11. The linked list benchmark (Azul).

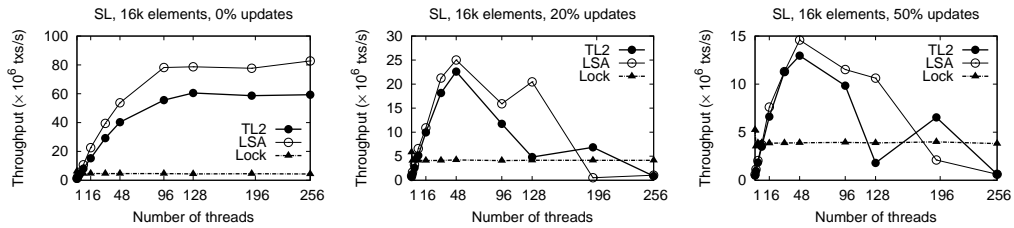


Fig. 12. The skiplist benchmark (Sun).

On the other hand we observe that the DEUCE STMs scale in an impressive way even with 20% updates (up to 32 threads). While DSTM2 shows no scalability. When we investigated deeper we found out that most of DSTM2 overhead rest in two areas. The most significant area is the contention manager which acts as a contention point, the second area is the reflection mechanism used by DSTM2 to retrieve and assign values during the transaction and in commit.

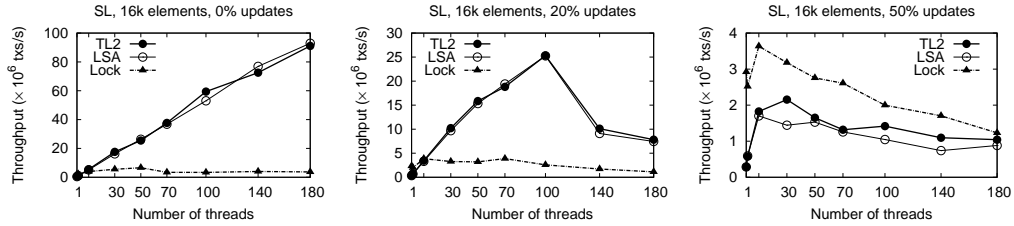


Fig. 13. The skip list benchmark (Azul).

Then we present the results of the tests with the linked list and the skip list on the Azul machine. The linked list benchmark is known to produce a large degree of contention as all threads must traverse the same long sequence of nodes to reach the target element. Results of experiments are shown in Figure 10 and Figure 11 for a linked list with 256 elements and different update rates. We observe that because there is little potential for parallelism, the single lock, which performs significantly less lock acquisition and tracking work than the STMs, wins in all three benchmarks. The STMs scale a bit until 30 threads when there are up to 5% updates, and then drop due to a rise in overhead with no benefit in parallelism. With 20% updates the max is reached at about 5 threads as after then we have statistically at least 2 concurrent updates and the STMs suffer from repeated transactional aborts.

Next, we consider results from the skip list benchmark. Skip lists allow significantly greater potential for parallelism than linked lists because different threads are likely to traverse the data structure following distinct paths. Results for a list with 16k elements are shown in Figure 12 and Figure 13. We observe that the STMs scale in an impressive way as long as there is an increase in the level of parallelism, and provide great scalability even with 20% updates as long as the abort rate remains at a reasonable level. We added a benchmark with 50% updates to show that there is a limit to their scalability, when we increase the fraction of updates to 50%, the STMs in Figure 13 reach a “melting point” much earlier (at about 10 threads versus 100 threads in the 20% benchmark) and overall the lock wins again because the abort rate is again high and the STMs pay with overhead without a gain in parallelism. We note that typically search structures have about 10% updates.

Real applications. Our last benchmarks demonstrate how simple it’s to replace the critical sections with transactions. The first takes a serial version of the Lee routing algorithm and demonstrates how a simple replacement of the critical sections by transactions significantly improves scalability. While the second takes a non-multi-threaded lock based a LRU cache implementation (Cache4J) and shows that it’s very simple to replace the critical sections but scalability isn’t promised.

Circuit routing is the process of automatically producing an interconnection between electronic components. Lee’s routing algorithm is attractive for parallelization since circuits (as shown in [2]) consist of thousands of routes, each of which can potentially be routed concurrently.

The graph in Figure 14 shows execution time (not throughput). As can be seen, DEUCE scales well, with the overall time decreasing even with a large board (MemBoard).

Cache4J is an LRU lock-based cache implementation. The implementation is based on two internal data structures, a tree and a hash-map. The tree manages the LRU while the hash-map holds the data. The Cache4J implementation is based

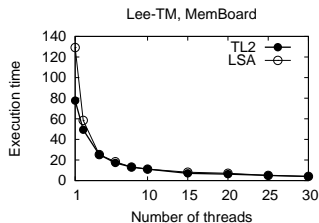


Fig. 14. The Lee Routing benchmark.

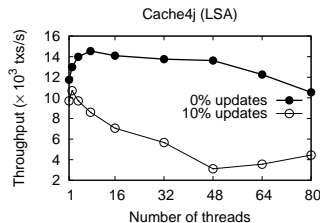


Fig. 15. The cache4j benchmark.

on a single global lock which naturally leads to zero scalability. The graph in Figure 15 shows the result of replacing the single lock with transactions. As can be seen DEUCE doesn't scale well, with the overall throughput slightly decreasing. A quick profiling shows that the fact that Cache4J is an LRU cache implies that every get operation also updated the internal tree. This alone makes every transaction an update transaction, the fact that the total throughput remains almost the same even with 80 threads encouraging due to the fact that transactions' biggest advantages, part of scalability, is code simplicity and code robustness.

In conclusion, DEUCE shows the typical performance patterns of an STM, good scalability when there is a potential for parallelism, and unimpressive performance when parallelism is low. The results, however, are very encouraging as we see that the fact that DEUCE is a pure Java library, with all the implied overheads, does not prevent it from showing scalability even for small numbers of threads.

7 Conclusion

We introduced DEUCE, a novel open-source Java framework for transactional memory. As we showed, DEUCE is the first efficient fully featured Java STM framework that can be added to an existing application without changes to its compiler or libraries. It's a proof that one can design an efficient pure Java STM without a compiler support.

Though much work obviously remains to be done, we believe DEUCE is ready for use by developers. It is freely downloadable from <http://code.google.com/p/deuce> under the Apache license.

In conclusion Table 2 shows a general overview comparing the two known Java STMs AtomJava¹ and DTMS2 vs DEUCE. This comparison shows that DEUCE gets the best from both worlds and provides new novel capabilities which yell much better usability and performance than currently exists.

Application	AtomJava	DSTM2	DEUCE
Locks	Object based	Object based	Field based
Instrumentation	Source	Runtime	Runtime
API	Non-Intrusive	Intrusive	Non-Intrusive
Libraries support	no-support	no-support	Runtime & offline support
Fields access	Anonymous classes	Reflection	low level Unsafe
Execution overhead	Medium	High	Medium
Extensible	Yes	Yes	Yes
Transaction context	Atomic block	Need to create a Callable	@Atomic annotated method

Table 2. AtomJava vs DSTM2 vs DEUCE

¹ We don't show comparison benchmark results with AtomJava due technical limitations and bugs found in its current implementation. Although small benchmarks showed DEUCE outperformed AtomJava

Acknowledgements. This paper was supported in part by grants from Sun Microsystems, Intel Corporation, Microsoft Inc., as well as a grant 06/1344 from the Israeli Science Foundation, European Union grant FP7-ICT-2007-1 (project VELOX), and Swiss National Foundation grant 200021-118043.

References

1. B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The jikes research virtual machine project: building an open-source research community. *IBM Syst. J.*, 44(2):399–417, 2005.
2. M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luján, and K. Jarvis. Lee-tm: A non-trivial benchmark suite for transactional memory. In *Proc. of the International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, pages 196–207, Berlin, Heidelberg, 2008. Springer-Verlag.
3. W. Binder, J. Hulaas, and P. Moret. Advanced java bytecode instrumentation. In *Proc. of the International Symposium on Principles and Practice of Programming in Java (PPPJ)*, pages 135–144, New York, NY, USA, 2007. ACM.
4. D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Proc. of the International Symposium on Distributed Computing (DISC)*, pages 194–208, 2006.
5. T. Harris and K. Fraser. Language support for lightweight transactions. In *Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 388–402, New York, NY, USA, 2003. ACM.
6. M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 253–262, New York, NY, USA, 2006. ACM.
7. M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proc. of the Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, New York, NY, USA, 2003. ACM.
8. B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *Proc. of the Workshop on Memory System Performance and Correctness (MSPC)*, pages 82–91, New York, NY, USA, 2006. ACM.
9. Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for C/C++. In *Proc. of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 195–212, New York, NY, USA, 2008. ACM.
10. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *Proc. of the International Conference on Compiler Construction (CC)*, pages 138–152, 2003.
11. W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.
12. T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proc. of the International Symposium on Distributed Computing (DISC)*, pages 284–298, Sep 2006.
13. T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *Proc. of the Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 221–228, New York, NY, USA, 2007. ACM.

Parallelizing Barnes-Hut Method on the Cell BE Architecture

B. Demiroz¹, H. Topcuoglu², M. Kandemir³ and O. Tosun¹

¹ Computer Engineering Department, Bogazici University, 34342, Istanbul, Turkey
{betul.demiroz,tosuno}@boun.edu.tr

² Computer Engineering Department, Marmara University, 34722, Istanbul, Turkey
haluk@marmara.edu.tr

³ Dept. of Computer Science and Engineering, Pennsylvania State University
University Park, PA 16802, USA
kandemir@cse.psu.edu

Abstract. The N-body (or many-body) problem is one of the crucial problems applied on large set of applications in various engineering and scientific domains. The N-body problem deals with n particles, each of which interacts with remaining ones during each time step. An exact formulation of this problem requires $O(n^2)$ computations; and several algorithms have been proposed to reduce the complexity of this problem due to large values of n . The main idea behind these algorithms is to group particles that are relatively close together and approximate them as a single particle in a spatial tree data structure. Among these algorithms, Barnes-Hut method is a popular one due to its simplicity and easily-programmable nature without requiring complicated data structures, with a computational complexity of $O(n \log n)$.

The Cell Broadband Engine Architecture is a heterogeneous multi-core chip architecture that has several unique features for high-performance computing. This paper presents a parallelization and empirical optimization scheme for the Barnes-Hut method on the Cell architecture, which is performed in three phases. In the first phase, the domain is partitioned into sub-domains which are assigned to synergistic processing element (SPE) by the power processing element (PPE). In the second phase, each SPE deals with its own particles in their sub-domains and construct the local trees of these sub-domains. When the local tree construction is completed, force acting on each particle is calculated by using all local trees constructed by SPEs. In the last phase, the total force acting on each particle is calculated, and positions and velocities of particles are updated. We implemented this method on both Cell BE architecture and Intel Xeon 3.0GHz. processor. The results show that the Cell BE architecture is significantly faster than Intel Xeon processor in various tests conducted with different number of particles.

1 Introduction

Almost all major chip manufacturers [1, 2] are working on chip multiprocessor (CMP) design and optimization, and one can expect that CMPs become the standard and the primary building blocks for personal desktop computers, embedded systems as well as large scale parallel machines. CMPs do sustain performance improvements by

increasing the number of cores on a chip with small form factor. The major design issues in the context of CMP architectures include improving performance, limiting power consumption and increasing the reliability of the architecture [4–9]. A CMP contains multiple CPUs (cores), an interconnection fabric, and some memory components, all packaged into a single chip. Since a CMP contains multiple CPUs, each CPU can be simpler and can be operated using low frequencies and supply voltages, as compared to single CPU based systems, thereby helping with both the power and reliability problem.

IBM Cell Architecture [1, 3, 10] is an example of heterogeneous multi-core architectures, jointly designed by IBM, Toshiba and Sony. IBM Cell engine contains a traditional microprocessor called the power processing element (PPE), eight synergistic processing elements (SPEs) that work in parallel, and an element interconnect bus. By using nine cores on a chip, IBM Cell offers a lot of computational power which makes it suitable for applications requiring high performance.

One of the computation intensive problems in the 13 dwarfs [12] which has not been studied on IBM Cell engine so far is the n-body problem, which is an important problem that can be applied to extensive applications from various domains in engineering and science. N-body problem studies the motion of particles by calculating pairwise interaction among n bodies for a predefined amount of time. At each time step, n-body problem requires $O(n^2)$ computations, which is not feasible since millions of particles are considered in most of the simulations. Therefore, many approximation algorithms are proposed in order to reduce the complexity of the problem. The Barnes-Hut method [15] is one of the most popular approximation algorithm, that requires a computational complexity of $O(n \log n)$. Its simplicity and ease of programability by using non-complex data structures improves the popularity of the Barnes-Hut method significantly.

This paper describes the design issues involved in mapping parallel implementation of the Barnes-Hut algorithm onto the Cell BE Architecture. In the first phase of our design, PPE assigns each SPE a portion of sub-domains, after the domain is partitioned into a set of sub-domains. Then, for every sub-domain, the local trees are constructed by all SPEs involved; and force on each particle is computed using the local trees constructed by SPEs. In the final phase, particles are distributed equally among SPEs and each SPE updates the position and the velocity of the particles assigned to them. Although, programming the Cell architecture is quite difficult because of explicit on-chip memory management, our results for the Barnes-Hut methods indicates that the required extra effort pays off. Specifically, Cell processor achieves a speedup of 6.22 with 8 SPEs compared to an Intel Xeon 3.0GHz. processor when 4096 particles are considered.

The rest of this paper is structured as follows. The next section discusses the main issues regarding the n-body problem and the Barnes-Hut method. The architectural features of the Cell processor are summarized in Section 3. Section 4 presents the details of our mapping of Barnes-Hut algorithm on the Cell processor. We present the results of our experimental evaluation in Section 5. Finally, Section 6 summarizes our main conclusions.

2 Barnes-Hut Method

The n-body problem simulates the evolution of a system containing n particles by calculating pairwise interactions among them. As a result of the interactions, a net force is exerted on each particle updating its velocity and position. This process is repeated for a predefined number of time steps. The force is dependent on the distance between particles; therefore, as the particles move, the forces need to be recalculated. Astrophysics, molecular dynamics, plasma physics, embedded SAR data processing and protein folding are some application areas of the n-body problem.

The n-body problem calculates all pairwise forces, therefore it requires n^2 interactions, at each time step. An exact formulation of this problem requires $O(n^2)$ computations and many methods have been proposed to reduce the complexity of this problem. The Barnes-Hut method [14, 15] is one of the popular algorithms due to its simplicity. Its computational complexity is $O(n \log n)$ and it uses a simple data structure which makes it easy to program. Another approximation algorithm is the Fast Multipole Method [13], which employs a complex data structure and has a complexity of $O(n)$.

The Barnes-Hut method is based on a hierarchical tree-based representation of space to group relatively close particles under a single tree node. This method has two phases, *tree construction phase* and *force computation phase*. In the tree construction phase, a tree representing the domain is constructed, which is used to cluster nearby bodies (i.e., particles) and to represent them with a single particle. Particles are added to the domain one by one. At each step, if the domain contains more than one particle, the domain is recursively divided into four equal sub-domains. This process finishes when there is a single element in each sub-domain. On the other hand, the net force of each particle is calculated in the force computation phase. For each particle, the nodes of the tree are traversed starting from the root node and the distance between the particle and the center of the node is calculated.

Each internal node stores an approximation of the particles it contains; the center of mass and the total mass of the particles. The nodes of the tree starting from the root are traversed for calculating the net force on each particle. If the center of mass of an internal node is sufficiently far away from the particle, the particles contained in the internal node are approximated as a single particle and the net force acting on the particle is calculated by using internal nodes mass and center of mass information. If the center of mass of an internal node is not sufficiently far away from the particle, then each of the internal nodes of the subtrees are traversed recursively.

In order to determine whether a node is sufficiently far away from a particle or not, the multipole acceptance criteria (MAC) [15] is computed. MAC is equal to the ratio of the dimension of the domain to the distance of the particle from the center of mass of the domain. If this ratio is less than a predefined constant λ , an interaction can be computed; otherwise, the node is expanded to its sub-nodes and the force between the particle and center of mass of the domain is computed recursively. The speed and the accuracy of the simulation is determined by setting a proper value for λ .

Although there are various parallel implementations of Barnes-Hut method in the literature [14, 17], which are either for message-passing architectures or shared-memory architectures; they can not be directly mapped to the Cell architecture. The sequential Barnes-Hut algorithm is given in Figure 1. This code is the non-recursive version of the Barnes-Hut implementations given in the literature; and it is the basis of our parallel implementation running on the Cell architecture.

```

For time  $\leftarrow$  0 to endTime do
  For i  $\leftarrow$  0 to particleNumber do
    Insert particle i to the tree
    Update center of mass and total mass of each internal node on the way
  End
  For i  $\leftarrow$  0 to particleNumber do
    Add root node to visitList
    While visitList is not empty
      Calculate distance between particle i and the center of mass of the node
      If center of mass of the node and particle is distant
        Calculate force acting on particle i
      Else
        Add children of node to visitList
      End
    End
  End
  For i  $\leftarrow$  0 to particleNumber do
    Update position and velocity of particles
  End
End

```

Fig. 1. Sequential Barnes-Hut Algorithm

3 The Cell BE Architecture

The Cell Broadband Engine [1, 3, 10] is a high performance architecture designed by Sony, Toshiba and IBM targeting multimedia and gaming applications. The Cell architecture is used in Sony's Play Station 3 gaming console, Mercury Computer System's dual Cell-based blade servers, IBM's Cell Blades (called QS20, QS21 and QS22). The Cell is designed to increase microprocessor efficiency in terms of both power and performance. The clock speed of the Cell processor is 3.2 GHz and it has a single-precision peak performance of 204.8 Gflops/s and double-precision peak performance of 14.6 Gflops/s.

The Cell consists of a traditional microprocessor (power processing element- PPE), 8 smaller and simpler processors (synergistic processing elements- SPEs) and an element interconnect bus (EIB) which connects the processors and provides access to main memory and I/O devices. The PPE is a dual-issue processor so it supports two-way simultaneous multithreading.

Each SPE consists of a Synergistic Processor Unit (SPU), a Local Store (LS) and a Memory Flow Controller (MFC). The instruction set of SPEs is designed to take advantage of 128-bit registers, and most of the instructions are SIMD instructions. Memory operations access 128-bits at a time even if the request data is 8-bits, so for an efficient implementation, the programmer should request 128-bits in each memory operation. SPEs are in-order processors with two instruction pipelines, namely the even pipeline and the odd pipeline. The even pipeline is responsible for arithmetic operations, and the odd pipeline deals with memory and branch instructions. In a single clock cycle, SPEs can dispatch two instructions if these instructions have no data dependency. SPEs do not have a cache, but they have a 256 Kbyte Local Store (LS) which can be called private memory. Both the program and the data should be in LS to be executed. SPEs have no direct access to main memory and a DMA controller is used to perform high bandwidth data transfers among the local store, main memory and other local stores.

The Element Interconnection Bus (EIB) connects all components of the CELL processor including the PPE, the SPEs, the main memory, and I/O. It supports a peak bandwidth of 204.8 Gbytes/s [11]. EIB is build of 16-byte wide four unidirectional rings, two in each direction.

4 Mapping and Optimizing the Barnes-Hut on the Cell Processor

In this section we describe how we parallelized the Sequential Barnes-Hut Algorithm given in Figure 1 on the synergistic processing elements (SPEs) of the Cell Architecture. As part of parallelization, we target to distribute data across SPEs. Since our workspace contains large number of particles, and each SPE contains a 256KB Local Store, it is not possible to hold the entire data and the tree on the SPEs. Therefore, the domain (i.e. the workload) is distributed equally among all SPEs in order to overcome both the limited size of the local stores and the synchronization barrier among SPEs.

In order to exploit the unique features of the Cell Architecture, a set of issues should be considered [16], like avoiding branches, vectorizing the corresponding code and overlapping memory access with computation. Since the IBM Cell Engine has no branch prediction mechanism, we consider the non-recursive version of the Barnes-Hut algorithm as the starting point. Some of the if-then-else statements in the code are replaced with *select bits* instruction for eliminating branches. Additionally, the loops in the code are unrolled partially by decreasing the number of iterations and replicating the instructions in the loops. In order to reduce memory access latencies, DMA transfers and computations are overlapped through double buffering. The Barnes-Hut method performs same computations on a large amount of 3D data repeating in each direction. In the force calculation phase, pairwise computations between particles and tree nodes are performed, where each computation is between a particle and the center of mass of a tree node. Each particle and the tree node are both represented with a vector; and all operations performed are SIMDized. Data transfers between memory and the local stores are for positions and masses of particles, so the particles in each sub-domain are stored as contiguous data in order to have a predictable memory access pattern. Double-buffering usage is efficient for this type of implementation. After both the force calculation and the update position phases are finished, the SPEs are synchronized by using mailboxes by sending and receiving 32 bit messages.

The parallel version of the Barnes-Hut algorithm for the Cell architecture can be expressed in three stages, which are *domain decomposition*, *tree construction and force calculation*, and *position and velocity updates*. The following subsections explain these stages in detail. The pseudo-codes of the Barnes-Hut algorithm for the PPE-specific and SPE-specific parts are given in Figure 2 and Figure 3, respectively.

4.1 Domain Decomposition

The domain is decomposed into smaller parts (called *sub – domains*) where each part contains limited number of particles that can fit in the LS of the SPE. Initially the PPE fetches the positions, velocities and masses of particles and it needs to know which sub-domain each particle belongs to, so it sends the position of particles equally to the SPEs and the SPEs decide on the sub-domain information. After the PPE gets the

```

Initialize Data Structures
Create SPEs
DMA: Inititate transfers to put blocks of particle coordinates to LS
Synchronization using mailbox
For time  $\leftarrow$  0 to endTime do
    Arrange all particles in the sub-domains
    Use load information of the sub-domains and distribute load between SPEs
    Map sub-domains to SPEs
    Wait for SPEs to finish tree construction and force calculation
    Synchronization using mailbox
    Wait for SPEs to finish position and velocity updates
    Synchronization using mailbox
End

```

Fig. 2. Barnes-Hut Algorithm: View within PPE

sub-domain information from the SPEs, it constructs sub-domains and uses the load information of sub-domains to give equal amount of workload to each SPE. In this stage, the domain is partitioned into sub-domains and the PPE assigns each SPE a set of sub-domains to work on. In the first iteration, the workload is divided into w equal units of data where total number of particles in the workload is equal to $w * p$. Here, p is the number of SPEs used in execution. When the first iteration ends, the number of nodes explored by the local tree representing a sub-domain during force calculation in the previous iteration is used as the workload metric of the sub-domain for the current iteration. After the PPE completes sub-domain allocation, it sends the SPEs the sub-domains that they will operate on and the SPEs fetch their data from memory and start their local tree construction.

4.2 Tree Construction and Force Calculation

In this phase of the algorithm, an octree representing the particles in a sub-domain is constructed by each SPE. Following this, the force acting on all particles in the workload is calculated on all SPEs using their own local trees. Since the SPEs have local stores of limited size, they operate on many sub-domains by getting each sub-domain one by one. When the local tree representation of the sub-domain is finished, it starts the force calculation among the particles and the sub-nodes by traversing the tree starting from the root.

Interactions with the top nodes of the tree are calculated first, and then the sub-nodes are explored if necessary. If the center of mass of an internal node is sufficiently far away from the particle, the particles contained in the internal node are approximated as a single particle and the net force acting on the particle is calculated by using internal nodes mass and center of mass information. To determine if a node is sufficiently far away from a particle, the multipole acceptance criteria [15] (MAC) is computed, which is the ratio of the dimension of the sub-domain to the distance of the particle from the center of mass of the given sub-domain. Therefore, if a particle is too distant from a cluster of particles (where MAC is less than λ), particles contained in the cluster are approximated using a single particle in an internal node. Each SPE is responsible of different sub-domains and there will be more interactions between particles and nodes

```

DMA: Initiate transfers to get blocks of particle coordinates
For i  $\leftarrow$  0 to particleNumber do
    Determine which sub-domain each particle belongs to
End
DMA: Put the data with its sub-domain information from LS back to main memory
Synchronize using mailbox
For time  $\leftarrow$  0 to endTime do
    While more sub-domains to visit
        While more blocks to process
            DMA: Load mass and coordinates of particles in the sub-domain from main memory into LS
            For i  $\leftarrow$  0 to subParticles do
                Insert particle i to the tree
                Update center of mass and total mass of each internal node on the way
            End
        End
        While more blocks to process
            DMA: Load mass and coordinates of all particles in the space
            DMA: Load force of all particles in the space
            For i  $\leftarrow$  0 to allParticles do
                Add root node to visitList
                While visitList is not empty
                    Calculate distance between particle i and the center of mass of the node
                    If center of mass of the node and particle is distant
                        Calculate force acting on particle i
                    Else
                        Add children of node to visitList
                    End
                End
            End
            DMA: Put force of all particles in the space
        End
        DMA: Put load information of sub-domain into MM
    End
    Synchronize using mailbox
    While more blocks to process
        For i  $\leftarrow$  0 to particleNumber do
            DMA: Get force exerted by other SPEs on particles
            Calculate total force acting on particles
            Update position and velocity of particles
            Determine which sub-domain each particle belongs to
        End
        DMA: Put new position, sub-domain and velocity of particles from LS to main memory
    End
    Synchronize using mailbox
End

```

Fig. 3. Barnes-Hut Algorithm: View within SPE

that are close to each other in the domain. As soon as the force calculation ends, each SPE writes back the calculated force values of all particles in its local tree to the memory and it gets the data of the new sub-domain. Tree construction and force calculation phase continues until all sub-domains assigned to the SPEs are processed.

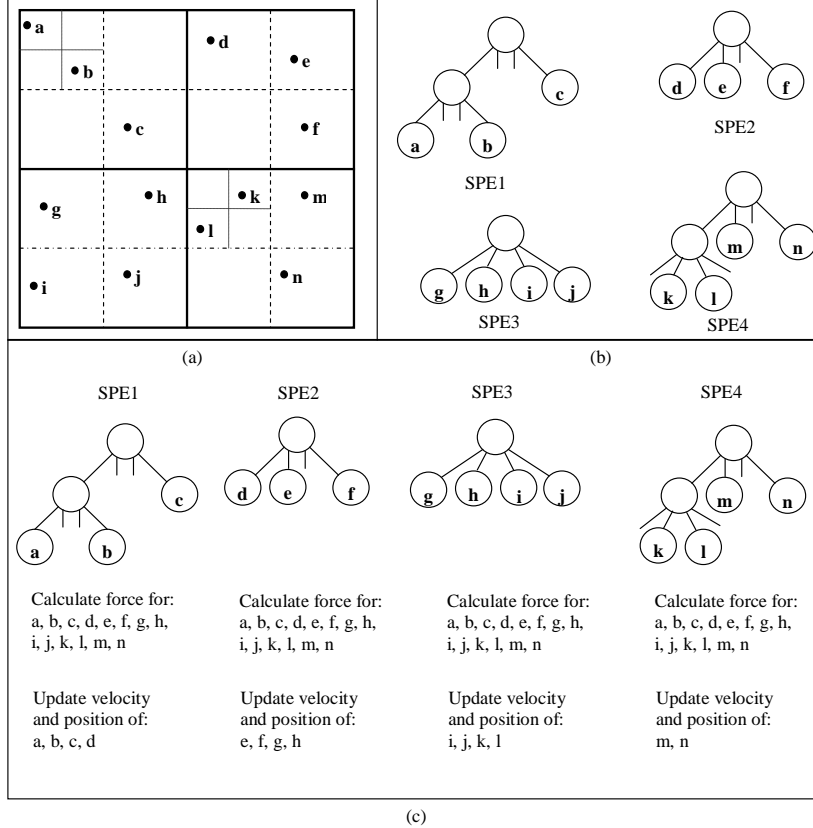


Fig. 4. A typical step of Barnes-Hut for 14 particles using 4 SPEs. (a) Domain Decomposition (b) Local Tree Construction in SPEs (c) Force Calculation and Position Updates in SPEs

Figure 4.b shows the local trees that are constructed by each SPE for the domain given in Figure 4.a. There are 14 particles given in the domain, where the particles are processed by 4 SPEs. In the first phase, the domain information of particles is calculated, and each SPE is assigned an equal number of particles to work on. Therefore, SPE1, SPE2 and SPE3 get 4 particles each and SPE4 gets the remaining 2 particles. In the second phase, sub-domains are distributed among SPEs. There are 4 sub-domains so each SPE deals with one sub-domain. According to the number of particles in the sub-domains, SPE1 and SPE2 gets 3 particles, SPE3 and SPE4 gets 4 particles. Although

SPE1 and SPE2 have the same number of particles, their tree construction times differ since the depths of their trees are not the same.

When the number of the nodes used to construct the local trees in Figure 4.b are compared, one can see that SPE1 and SPE3 have 5 nodes, SPE2 has 4 nodes and SPE4 has 6 nodes. Therefore, SPE2 has the lowest tree construction time and SPE4 has the highest tree construction time. It should be noted that SPE1 and SPE2 differ in terms of the tree construction time, although they have equal number of particles in their sub-domains. Figure 4.c shows the force calculation phase, the local tree is traveled 14 times, once for each particle, and as a result, the time it takes for calculating the force is much longer than that of tree construction. Consequently, this phase results in an unbalanced load among SPEs. If there are more number of local trees of SPEs unbalanced, SPEs will wait longer at the synchronization barrier.

4.3 Position and Velocity Updates

After the force calculation phase is over, the net force acting on each particle is calculated by using all force values calculated on all local trees. Then, the particle's velocity and position values are updated accordingly. This process is repeated for a predefined number of iterations; and at the end of the simulation, each particle has new position and velocity values.

5 Experimental Analysis

The performance results presented in this section are from actual runs on an IBM Blade Center QS20 with two 3.2GHz Cell BE processors, 512 KB Level 2 cache per processor and 1 GB memory. Only one of the processors of QS20 is considered in our experimental study. The code is compiled using gcc compiler in the Cell SDK 3.0 with -O3 optimization flag for performance analysis. As mentioned before, loop unrolling, double-buffering and vectorization are used to decrease execution time on the SPEs. The code running on the PPE is also optimized by loop unrolling, branch elimination and vectorization. In all the experiments, the number of particles used vary from 1024 to 32768, the number of buffers used is 2, λ is 0.5 and the number of iterations is 50 unless stated otherwise.

In the first set of tests, the performance of our algorithm is measured by varying the number of the SPEs used for different particle sizes. Figure 5 shows the speedup values that are normalized to a single SPE. Running time of the PPE is also included in the experiments. Considering that the Barnes-Hut algorithm is branchy, the PPE runs the code faster than a single SPE. When 1024 particles are considered, 2 SPEs run 1.61, 4 SPEs run 2.2 and 8 SPEs run 3.98 times faster than 1 SPE. 2 SPEs complete 1.95, 4 SPEs complete 4.18 and 8 SPEs complete 8.45 times faster when 32768 particles are considered. This shows that application scales well as the number of the SPEs increase. The performance of the SPEs with respect to CPI values are also measured by using IBM Cell Simulator. In the tree construction phase, the CPI values range between 1.69 and 1.71, and in the force calculation phase, they are between 1.72 and 1.79. The CPI values are higher than the theoretical CPI value of 0.5 which is due to the branchy nature of the algorithm since in both phases the tree is traversed for each particle. Communication and computation time of the application are also measured by using

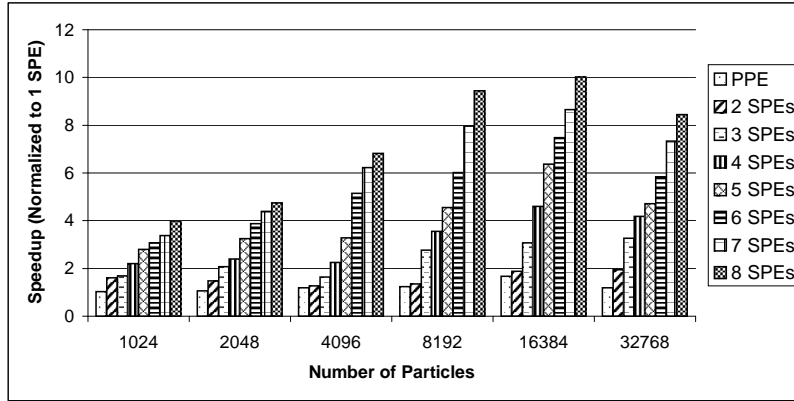


Fig. 5. Speedup with respect to different number of SPEs and PPE

IBM Cell Simulator. For a sample run, the force calculation phase takes 87.3% of the total execution time, whereas the tree construction phase takes 2.8%, and position updates takes 1.1% of the total execution time. Synchronization time among SPEs takes up to 7.8% of the total execution time. The DMA operations take 1% of total execution time due to double-buffering.

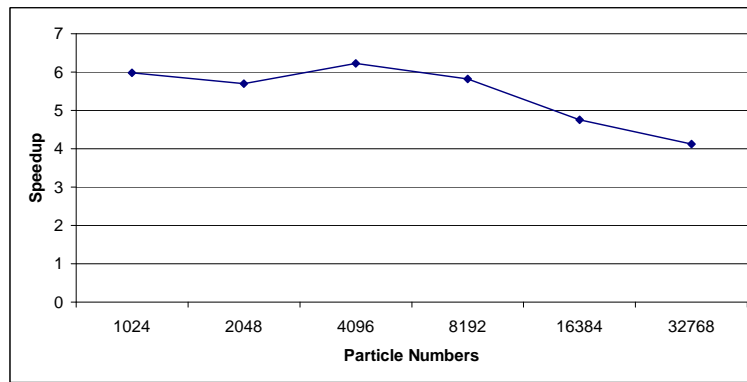


Fig. 6. Speedup of 8 SPEs relative to the Intel Xeon with respect to different number of particles

In the second set of experiments, we present a performance comparison of our implementation on the Cell processor against Dual Core Intel Xeon processor with 3.0 GHz clock frequency. Barnes-Hut code on Intel Xeon is compiled with gcc version 3.4.6 and -O5 optimization flag. The optimizations on the application code, except parallelization, vectorization and Cell specific optimizations, are applied to both architectures for a fair comparison. Additionally, the SPEs can construct trees of domains with limited number of particles since they have LS limitations, and this increases number of nodes explored in the force calculation phase whereas Intel Xeon has no such memory limitations so all particles are considered at once in the tree construction phase, which

improves performance of the application. The domain decomposition and workload distribution are accomplished by the PPE and the SPEs have to synchronize with each other after the force calculation and the position update phases are finished. The SPEs have a limited local store and can only construct a tree consisting of at most 370 nodes, so a single sub-domain used for tree generation contains at most 208 particles. To ease workload distribution and load balancing, the workspace is initially divided into 512 sub-domains and in each tree generation step, the SPEs are assigned N number of sub-domains, where the total number of particles in N sub-domains is less than or equal to 208. As the number of particles increases, all nearby bodies may not be clustered in a single tree because of space limitations, and this increases the number of nodes explored in the force calculation step.

Figure 6 compares the speedup among 8 SPEs and Intel Xeon with respect to different number of particles. When 4096 number of particles are considered, the Cell/B.E. outperforms Intel Xeon with a speedup of 6.22. As the number of particles increases, we cannot achieve an ideal speedup since there is an increase in the total number of local trees generated in a single step. As a result, all of the internal nodes containing cluster information cannot be created, consequently, the number of clusters used in the force calculation phase decreases. When less number of clusters are considered, the number of nodes explored as well as the number of comparisons made increases, which decreases the performance of the system.

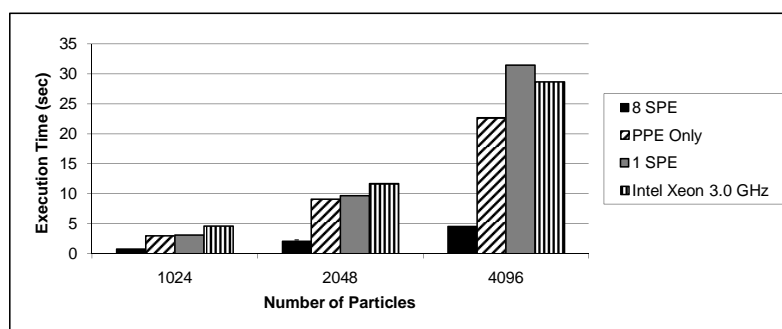


Fig. 7. Performance Comparison with respect to number of particles given in the range [1024-4096]

Figure 7 and Figure 8 compare the running time of our algorithm on both Cell processor with 1 SPE, 8 SPEs, the PPE with an Intel Xeon 3.0GHz. processor by varying the number of particles in the domain. If we have a closer look at Figure 7, the best performance is obtained when 8 SPEs are used. For 1024 and 2048 particles, Intel Xeon shows the worst performance. When the performance of 4096 particles are compared, 1 SPE shows the worst performance and the PPE only case outperforms both Intel Xeon and 1 SPE. As the number of particles increases, the performance of Intel Xeon increases and it outperforms both PPE only case and 1 SPE.

Recall that the MAC criteria calculates the ratio of the distance between the center of mass of the node and a particle to the dimension of the sub-domain. The term λ is used to decide whether to expand the corresponding node or to represent all particles

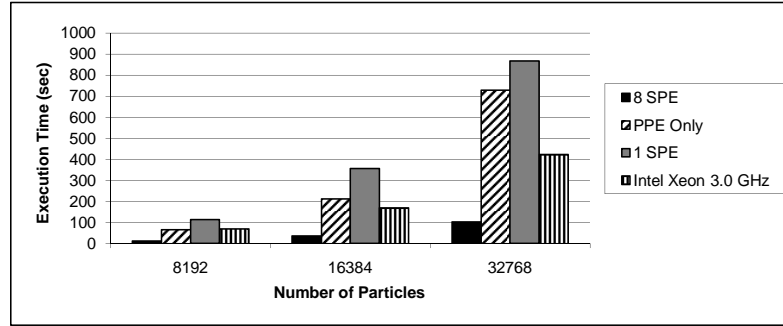


Fig. 8. Performance Comparison with respect to number of particles given in the range [8192-32768]

that belong to the node with the center of mass of the node. The value of λ is in the given range $0 \leq \lambda \leq 1.0$; and if it is equal to 0, the Barnes-Hut application is turned into the N-Body problem where all nodes are explored and pairwise force among all particles are calculated, in this case clustering of nearby particles has no effect. Therefore, the increase in λ value decreases the number of nodes explored and it also decreases the accuracy of the calculations made. Since the application spends most of its time doing force calculation, the execution time decreases as the number of nodes explored decreases because the number of pairwise force calculation decreases.

Figure 9 compares the performance of the Cell processor of 8 SPEs with Intel Xeon for different λ values. When λ value is close to 1, it does not use the individual positions and mass values of the particles, but uses mostly cluster information of upper level nodes in the force calculation phase. If the SPEs do not contain the upper level cluster information because of space limitations, then they use lower level cluster information but this increases the number of comparisons made and this explains the reason why speedup decreases when λ is 0.7 and 0.9. When λ value is close to 0, cluster information of the lower level nodes are more frequently used, and these are mostly available in the local trees of the the SPEs, so as a result, the performance of the application increases. The best performance with a speedup of 7.83 is obtained when λ value is 0.3 and 4096 particles are considered.

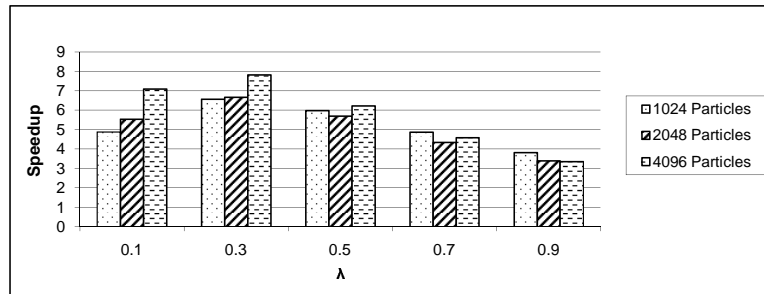


Fig. 9. Speedup of 8 SPEs relative to the Intel Xeon with respect to different λ values

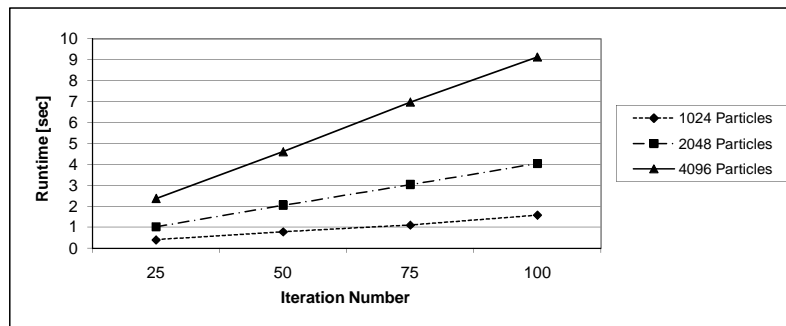


Fig. 10. Performance with respect to different number of iterations

In the last set of experiments, the performance of our algorithm is measured by varying the number of iterations on 8 SPEs. (see Figure 10) There is a linear increase between the number of iterations and the execution time and this shows that each iteration takes nearly the same amount of time.

6 Conclusions

In this paper, we present and evaluate a parallel implementation of the Barnes-Hut method on the Cell architecture. Memory requirement of this application is very high, since both local tree representing the domain and the positions and masses of the particles should be stored in the Local Stores of the SPEs at the same time. Consequently, an efficient workload (i.e., domain) distribution on the SPEs is needed. Also the domain needs to be divided into sub-domains to overcome memory needs and to balance load. The parallel version of the Barnes-Hut method presented in this paper contains three stages. In the first phase, the domain is decomposed into smaller sub-domains and the sub-domains are assigned to the SPEs. In the second phase, tree construction and force calculation in each sub-domain is performed for sub-domains. Finally the velocities and positions of the particles are updated. Compared to many other parallel Barnes-Hut implementations, our proposed work combines the tree construction and force calculation phases and this phase is repeated for the number of sub-domains assigned to the SPEs. In the experimental study, the performance of the application is explored by using different number of SPEs, different λ values, and different number of particles. While programming the Cell architecture is more difficult than that of Intel Xeon (due to explicit on-chip memory management), our results show that the required extra effort pays off. That is, the Cell version performs much better than the sequential version running on Intel Xeon processor.

Acknowledgments

This research was supported by The Scientific and Technological Research Council of Turkey (TUBITAK) with a research grant (Project Number: 108E035). Additionally, part of this research has been funded by Bogazici University Research Fund -08HA101D. The authors acknowledge Georgia Institute of Technology, its Sony-Toshiba-IBM Center of Competence, and the National Science Foundation, for the use of Cell Broadband Engine resources that have contributed to this research.

References

1. J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy, *Introduction to the Cell Multiprocessor*, IBM Journal of Research and Development, Vol. 49, Issue: 4-5, pp.589-604, July 2005.
2. P. Kongetira, K. Aingaran, and K. Olukotun, *Niagara: A 32-Way Multithreaded SPARC Processor*, IEEE MICRO Magazine, Vol. 25, Issue: 2, pp. 21-29, April 2005.
3. H.P. Hofstee, *Power Efficient Processor Architecture and The Cell Processor*, In Proceedings of the International Symposium on High Performance Computer Architecture, pp. 258-262, 2005.
4. K. Olukotun and L. Hammond, *The Future of Microprocessors*, ACM Queue, Vol. 3, Issue:7, pp. 26-29, September 2005.
5. R. Kumar, D. Tullsen and N. Jouppi, *Heterogeneous Chip Multiprocessors*, IEEE Computer Society, Vol. 38, No. 11, pp. 32-38, November 2005.
6. M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffmann, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpfen, M. Frank, S. Amarasinghe, and A. Agarwal, *The RAW microprocessor: A computational fabric for software circuits and general purpose programs*, IEEE Micro, Vol. 22, Issue:2, pp. 25-35, March 2002.
7. K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore, *Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture*, In Proceedings of the 30th Annual International Symposium on Computer Architecture, pp. 442-433, 2003.
8. S. Swanson, K. Michelson, A. Schwerin, and M. Oskin, *Wavescalar*, In Proceedings of the annual IEEE/ACM International Symposium on Microarchitecture, pp. 291-302, Washington, DC, USA, December 2003.
9. K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, *Smart memories: a modular reconfigurable architecture*, In Proceedings of the annual International Symposium on Computer Architecture, pp. 161-171, New York, NY, USA, 2000.
10. D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki and K. Yazawa, *The design and implementation of a first-generation Cell processor*, In International Solid State Circuits Conference, San Fransisco, February 2005.
11. M. Kistler, M. Perrone, F. Petrini, *Cell Multiprocessor Communication Network: Built for Speed. Communication Build for Speed*, IEEE Micro Vol. 26, No. 3, pp. 10-23, 2006.
12. K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, K. Yelik, *The Landscape of Parallel Computing Research: A View from Berkley*, Technical Report, EECS Department, University of California at Berkley, UCB/EECS-2006-183, December, 2006.
13. L. Greengard, *The rapid evolution of potential fields in particle systems*, Cambridge: The MIT press, 1988.
14. A. Grama, V. Kumar, and A. Sameh, *Scalable Parallel Formulations of the Barnes-Hut Method for N-Body Simulations*, In Proceedings of Supercomputing, pp. 439-448, 1994.
15. J. Barnes and P. Hut, *A Hierarchical $O(n \log n)$ Force Calculation Algorithm*, Nature, vol.324, pp. 446-449, December 1986.
16. D. A. Brokenshire, *Maximizing the Power of the Cell Broadband Engine Processor: 25 Tips to Optimal Application Performance*, IBM developer Works, June 2006. .
17. U. Ramachandran et. al, *Architectural Mechanisms for Explicit Communication in Shared Memory Multiprocessors*, In Proceedings of Supercomputing, 1995.

Expressing Inter-task Dependencies between Parallel Stencil Operations

Per Larsen, Sven Karlsson and Jan Madsen

DTU Informatics
Technical University of Denmark
{pl, ska, jan}@imm.dtu.dk

Abstract. Complex embedded systems are designed under tight constraints on response time, resource usage and cost. Design space exploration tools help designers map and schedule embedded software to complex architectures such as heterogeneous MPSoC's. Task graphs are coarse grained representations of parallel program behaviour which are used to evaluate the feasibility of a particular design. However, automatically extracting an accurate task graph from source code is challenging. This paper investigates how to describe data dependencies to aid tools based on program analysis in extracting task graphs from source code. We will examine a common parallel programming pattern – stencil operations – and show that even for such codes with a regular control flow, the precise dependencies between two stencil operations cannot always be determined at compile time.

We introduce a language construct which i) captures an upper bound on the number of dependencies between successive stencil operations and ii) instructs the compiler to generate code which ensures that the bound holds for each execution of the program.

The impact of our proposal is evaluated using a micro-benchmark and two soft real-time embedded image processing applications. The coding effort is low – at most one line of code per parallel loop was added. The performance impact is evaluated on a quad-core Linux workstation and we observe no statistically significant slowdown.

1 Introduction

Today's embedded systems are expected to process complex data in real time while being highly energy efficient and inexpensive to manufacture. The aforementioned goals are often in conflict so design decisions must be made. Design space exploration, DSE, tools can help designers make well-informed decisions by automatically finding and evaluating the trade-offs between the set of feasible designs [1,2,3].

To evaluate each design point, a coarse grained model of the application behavior must be extracted from its source code. Task graphs, which are directed acyclic graphs, are commonly used for this purpose. Each node in the graph represents sequential computation and each edge represents a precedence constraint

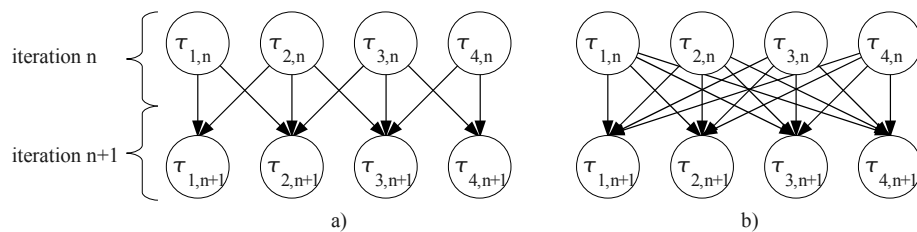


Fig. 1: a) Desired task graph for two iterations of heat diffusion example on four threads. b) Task graph fragment with over-approximation of inter-task dependencies.

between two tasks due to a data or control flow dependence in the source code [4].

Neither program analysis nor traces of program execution alone are sufficient to extract task graphs from source code. The former approach [5,6,7] computes a safe over-approximation of the inter-task dependencies for all program executions. Trace based approaches [8,9] can compute the precise dependencies between the tasks for a single program execution. However, trace based task graphs do not contain all possible dependencies since traces can only be generated for a tiny fraction of all possible program inputs.

In this paper, we show that it is possible to avoid some over-approximated edges and thus increase the accuracy of task graphs computed using program analysis by adding extra information to the source code through a programmer inserted directive.

Listing 1.1 contains a simple heat diffusion program parallelized with OpenMP [10] directives. It repeatedly applies a non-compact stencil which is five elements wide to a one dimensional array. The number of iterations in the inner loop and the number of threads are unknown. Stencil widths can also be unknown. The goal is to determine the dependencies between successive iterations of the *outer* loop.

Listing 1.1: Heat diffusion example. The code is adapted from Mattson et al. [11]

```

1 for(int k = 0; k < NSTEPS; ++k) {
2   /* instruct compiler to distribute iterations among available threads */
3   #pragma omp parallel for schedule(static)
4   for(int i = 2; i < NX-2; ++i) {
5     /* compute new value in 'ukp1' based on neighbouring values in 'uk' */
6     ukp1[i] = uk[i] + (dt/(dx*dx)) * (-1/12*uk[i-2]
7     +4/3*uk[i-1]-5/2*uk[i]+4/3*uk[i+1]-1/12*uk[i+2]);
8   }
9   /* swap pointers before executing next iteration */
10  tmp = ukp1; ukp1 = uk; uk = tmp;
11 }

```

Figure 1a shows a fragment of the task graph corresponding to two successive iterations of the outer loop *assuming* the number of iterations is at least twice the number of threads. Program analysis, however, cannot prove that each thread will receive more than one iteration of the inner loop – as thread and iteration counts are unavailable – and must therefore produce the task graph fragment shown in Fig. 1b.

Scope and Contributions We allow ourselves to assume that programs i) are parallelized with OpenMP ii) do not contain data races and iii) access arrays and dynamically allocated memory correctly.

This paper makes the following contributions:

- We introduce the `taskshare` directive (Section 3) which is necessary to derive task dependencies between parallel stencil operations when stencils wider than three elements are used.
- We also present the required runtime support (Section 4) so that object code generated by a compiler can check at runtime that the actual dependencies match the claims of the programmer inserted `taskshare` directives.
- The impact of the new directive is evaluated in terms of the resulting task graphs, coding effort and performance (Section 5). We use the heat diffusion example and two soft real-time embedded programs.

The ability of the directive to exclude dependencies otherwise reported by program analysis is directly proportional to the stencil width. When stencils spanning five elements are used, it enables a 40% reduction of dependencies between task pairs while stencils only three elements wide do not benefit from the directive. The dependencies excluded by the directive can allow DSE tools to find more feasible designs and prevent over-provisioning of resources. Secondly, less dependencies lowers the running times of task scheduling algorithms whose asymptotic complexity increase with the number of dependencies in the task graph [12,13,14,15].

In terms of coding effort, the impact is low – the `taskshare` directive adds at most one additional line of code to each loop performing a stencil operation. In terms of performance, benchmark measurements of three applications show no statistically significant impact of runtime checks at the 95% confidence level.

2 Related Work

Vallerio et al. proposed an automated task graph extraction approach from unmodified C code [7]. Explicit parallelism is not supported and the extraction method relies on very conservative assumptions about the use of pointers and arrays which lead to task graphs containing dependencies which do not exist in the actual program. The challenge presented by pointer aliasing in context of task graph extraction was addressed in our previous work [16].

Adve and Sakellariou address task graph extraction in the context of high-performance computing [5]. They assume that a distributed memory programming model, MPI [17], is used exclusively for communication among tasks thereby making dependencies explicit. They also argue in favor of generating task graphs from OpenMP programs.

Ha has proposed the HOPES programming environment for development and mapping of embedded software to MPSoC's [18]. OpenMP is used to express data-parallelism in programs while task-parallelism is expressed in a synchronous data flow model. The difficulties of calculating dependencies among tasks representing parallel loops are not considered.

Liu and Dick present a tool which can generate communication graphs by tracing loads and stores during execution rather than compile time analysis [8]. Unlike task graphs, communication graphs cannot be parameterized or composed hierarchically to analyze application behavior across different program executions and execution platforms.

The hArtes project aims to develop an end-to-end development framework for real-time embedded systems [19]. The approach to task graph extraction is based on automatic parallelization of sequential C code. However, only a few types of code can be parallelized automatically. Secondly, being able to derive a parallel program from a sequential program does not imply that an accurate task graph can be derived from the program. As this paper will explain, the number of dependencies between two parallel stencil operations depends on factors which are seldom available at compile time.

Schmitz et al. [1] determine inter-task dependencies by hand to schedule and map small applications to a smartphone. This approach is labour intensive and prone to errors.

3 Directive to Quantify Sharing among Tasks

We propose a new directive which can assert the maximal number of threads which will access the same array slice when executing a parallel loop.

The syntax of the directive in the informal notation of OpenMP [10] is: `#pragma depends [taskshare(expr, ts) ...]` where *expr* must evaluate to either an array or a pointer to an array and *ts* must be the maximal number of threads concurrently accessing a slice of *expr*. The directive can only be used with an `omp for` directive and must appear directly before it. If several loops in a loop nest are parallelized with `omp for` then each should have its own `taskshare` directive. Finally, annotated loops must be coded such that the width of the stencil can be determined at compile time. This requirement is justified in Sect. 3.

The `taskshare` directive is useful when a sequence of stencil operations are encountered during task graph extraction in preparation for design space exploration. Program analysis can use the second argument of the `taskshare` directive to determine the number of dependencies from tasks belonging to stencil operation *n* to each task in stencil operation *n* + 1 instead of using a conservative approximation. In the heat diffusion example, the dependencies excluded are those which are in Fig. 1b but not Fig. 1a.

Correctness of the Directive At runtime, the iterations of a parallel loop are divided into a number of slices which in turn are mapped to threads. Consider a situation where the inner loop in the heat diffusion example contains 24 iterations which are divided into four slices of length six such that thread 0 executes iterations 0-5, thread 1 executes iterations 6-11, etc. Since the stencil width is five, thread 1 will read elements 4-13 which were written in the previous iteration of the outer loop – thread 0 wrote elements 4-5, thread 1 wrote 6-11

Listing 1.2: Innermost loop in the example with `taskshare` directive applied. The directive is underlined.

```

1 #pragma depends taskshare(uk,3)
2 #pragma omp for schedule(static)
3 for(int i = 2; i < NX-2; ++i) {
4     ukp1[i] = uk[i] + (dt/(dx*dx)) * (-1/12*uk[i-2]
5         +4/3*uk[i-1]-5/2*uk[i]+4/3*uk[i+1]-1/12*uk[i+2]);
6 }
```

and thread 2 wrote 12-13. The task executed by thread 1 therefore depends on three of its predecessor tasks. Generally the number of predecessors of a task equals the number of adjacent slices read plus one.

The number of adjacent array slices read by a task depends on the width of the stencil and the length of the slices – the wider the stencil and the shorter the length of each slice, the more adjacent slices are read. To check the correctness of the `taskshare` directive, the stencil width and minimal slice length must be computed to check that the number of adjacent slices read plus one is less than or equal to the value of `ts` argument in the directive. We note that the directive is not needed in case the stencil width is three because each task then reads at most three array slices regardless of their length.

Listing 1.2, shows the `taskshare` directive applied to the inner loop in the heat diffusion code. The `schedule` clause [10, p. 43] controls how iterations are divided into slices and mapped to threads. A certain slice length can be requested by the *chunk size* parameter. Since the schedule is `static` with no chunk size, each thread receives a slice with a length in proportion to the total number of iterations. The slice length is therefore greater than $\max(\lfloor \frac{n}{t} \rfloor, 1)$ for n iterations and t threads. Whenever $2n \geq t$, the length of each slice is at least 2 and since a five-point stencil can never access more than two elements of an adjacent slice, it accesses one adjacent slice on each side. Thus, in cases where the schedule is `static`, the stencil is five elements wide and the number of iterations is at least twice as great as the number of threads then the inserted directive correctly bounds the number of threads which concurrently access each slice in the loop.

Prerequisites for Runtime Checks The correctness of each `taskshare` annotation should be checked for correctness to guard against human error. Changes to i) the loop schedule or chunk size ii) the number of iterations iii) the stencil width or iiiii) the number of threads can render the assertion captured by the directive invalid. None of these factors are required to be compile time constants but must be loop invariant [10] so checks must be performed at runtime.

To insert checks, the variables containing the values of the four factors must be identifiable by the compiler. The variables controlling loop schedule, iteration count and chunk size are trivially obtained. However, the width of the stencil is, in some cases, hard to identify at compile time.

Loops annotated with the `taskshare` directive must therefore adhere to three coding rules.

1. Do not convert multi-dimensional arrays to one-dimensional arrays.
2. Use array indexing expressions rather than pointer arithmetic.
3. Two `taskshare` clauses must not contain conflicting information on the number of threads accessing a possibly aliased array slice.

The first two rules ensure that program analysis can distinguish between accesses in different dimensions of a multi-dimensional array. The third criteria addresses pointer-aliasing. A `taskshare` clause specifies the number of threads that access an array slice. Aliasing may occur since the directive allows for the use of pointers to specify the array.

4 Runtime Checks

To measure the performance impact of the runtime checks, a library containing the necessary functionality was implemented. We also wrote a compiler plug-in which inserts runtime checks of `taskshare` directives during compilation. We used `llvm-gcc` version 2.5 with and `gcc` 4.2.1 which includes the `libgomp` OpenMP runtime. The exact nature of the runtime checks depends on how an OpenMP runtime implements the four different loop schedules. Our work is specific to `libgomp`.

If the schedule type of the parallel loop is either `static` or `dynamic` then the minimum slice length can be computed in constant time when the loop begins executing. However, if the schedule is `guided` then the length of each slice depends on all previously computed slices in the loop. The shortest slice is therefore found by comparing each slice as it is computed during the execution of the parallel loop – thus the overhead is in proportion to the number of slices. The `runtime` schedule is handled similarly to `guided`.

In case a runtime check detects a that a directive was not satisfied, it raises a runtime error. This implies that a task graph based on the assertions in `taskshare` directives does not match observable program behavior. It is left to the program to determine whether to continue execution or not. Runtime errors can be avoided either by modifying the `schedule` clause to lengthen the shortest slice, by modifying the directive, i.e. increasing the bound on threads concurrently accessing an array slice, or decreasing the stencil width if possible.

Inserting Runtime Checks during Compilation Figure 3 illustrates how the compiler transforms code annotated with an `omp for` directive.

The loop with a `static` schedule and unspecified chunk size in Fig. 3a is transformed into Fig. 3b whereas all other types of loops corresponding to Fig. 3c are transformed to Fig. 3d. In the first case, each thread receives its slice of the iteration space by means of a single call to `loop_start`. In the second case, each thread receives the initial slice from a call to `loop_start` and subsequent slices by calling `loop_next` at the end of the compiler inserted loop which encapsulates the original loop.

Our `llvm-gcc` plug-in adds an additional compilation step after the processing of OpenMP directives as illustrated in Fig. 2. In this step, the code generated

```

1 #pragma omp for schedule(static)          1 #pragma omp for schedule(...)
2 for(i=lb;i<ub;i+=incr)                  2 for(i=lb;i<ub;i+=incr)
3 /* <loop body> */                       3 /* <loop body> */

```

a) Parallel loop with `static` schedule and unspecified chunk size.

```

1 /* slice begin, slice end */            1 /* slice begin, slice end */
2 long sb, se;                            2 long sb, se;
3 if(loop_start(                          3 if(loop_start(
4   lb,ub,incr,"static",0,&sb,&se) {       4   lb,ub,incr,ls,c,&sb,&se)) {
5   for(i=sb,i<se;i+=incr)                5   do {
6   /* <loop body> */                     6     for(i=sb,i<se;i+=incr)
7   }                                       7     /* <loop body> */
8 }                                       8   }
9 loop_end();                             9   while(loop_next(&sb,&se));
                                         10  }
                                         11 loop_end();

```

b) Code generated for static schedule and default chunk size.

c) All other schedule choices.

d) Code generated for all other schedule choices.

Fig. 3: a-b) Translation of loop with `omp for` directive with `schedule(static)` clause into parallel loop. c-d) Translation of loop with `omp for` directive for all other `schedule` choices. Functions which may be replaced by wrapper functions containing runtime checks are underlined.

from the `omp for` directives is identified and runtime checks are inserted for loops annotated with the `taskshare` directive. When the minimal slice length can be calculated before entering the loop, i.e. when schedule type is `static` or `dynamic`, the call to `loop_start` is replaced by a call to `loop_start_wrapper`. The call performs a runtime check and then forwards the call to `loop_start`. Otherwise the call to `loop_next` is replaced with a call to `loop_next_wrapper`, which performs a runtime check on the length of the slice which was just processed and forwards the call to `loop_next`.

5 Experimental Results

We evaluate the required programming effort and how the runtime checks impact execution time of three parallel OpenMP benchmarks. We also evaluate the impact of the `taskshare` directive in terms of its ability to exclude dependencies between tasks which would otherwise have been computed by program analysis. This is only done on the first benchmark, however, as we have not been able to

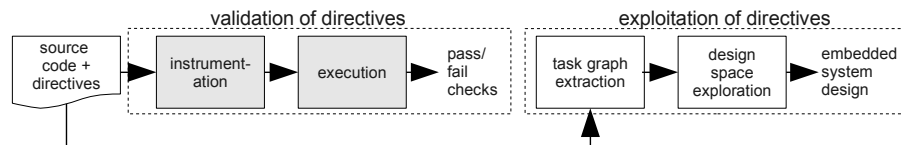


Fig. 2: Use of the `taskshare` directive in an embedded system design flow. Directives are validated by instrumenting the program and executing it to determine if all checks pass for all relevant program inputs. The information in the directives can then be exploited by a task graph extraction tool in preparation for the design space exploration step.

Name	Sequential time	Relative speed-up		Memory use
		2 threads	4 threads	
Heat diffusion	1.3781	1.9	3.1	2
Demosaicing	0.8980	1.9	3.5	221
Edge detection	0.5971	1.9	3.5	192

Table 1: Memory usage and speed-up of the benchmark applications run with `static` scheduling of loops. All times are in seconds and memory consumption is in megabytes.

obtain real embedded code using stencils wider than three elements. The embedded code can still be used to evaluate the runtime checks since the overhead of each check is independent of the stencil width.

The average execution time is calculated from thirty consecutive executions. We compare these averages with a two-sided, unpaired t-test using 95% confidence intervals. To quantify the utility of the directive, we study the relative change in the number of dependencies between a pair of tasks belonging to different stencil operations - e.g. tasks $\tau_{2,n}$ and $\tau_{2,n+1}$ in Fig. 1. Finally, the number of directives added in the source code is used to approximate the required programming effort.

5.1 Experimental Set-up

All experiments were performed on a workstation with a quad-core 2.66 GHz Intel Core i7 920 CPU and 3 GB DDR3 RAM. It had 256 KB L2 cache per core and 8 MB shared L3 cache. The operating system was 32-bit Ubuntu Linux 9.04 with kernel version 2.6.28. The measurements we will present were obtained using four threads but similar results were observed for experiments with one, two and eight threads.

Only the parts of the benchmark performing parallel work and which can contain our runtime checks are included in the execution time. The scalability of the parallelized parts of the benchmark applications is shown in Table 1.

We used the timing-facilities included in OpenMP which, on our platform, uses hardware cycle counters and obtains precision in the nanosecond range. To reduce the variability between executions we disabled dynamic power and frequency scaling, all hardware pre-fetching and simultaneous multi-threading (Hyper-Threading) via BIOS settings. The system ran in single user mode to reduce interference from background processes.

We have compared the quality of the code generated by `llvm-gcc` and `gcc`. This was done to ensure that the use of the LLVM optimizer did not do a poor job thus lowering the impact of our non-optimizable instrumentation. Our experiments show that that code generated by `llvm-gcc` is consistently faster than that of `gcc`. We used the optimization flag `-O2` which is supported by both compilers for all experiments.

5.2 Heat Diffusion

The heat diffusion code was used as a micro-benchmark. An `omp parallel` directive was put before the outer loop and an `omp for` replaced the `omp parallel for` to improve efficiency. It executes the same stencil operation iteratively for a number of time-steps so the inserted runtime checks are exercised repeatedly.

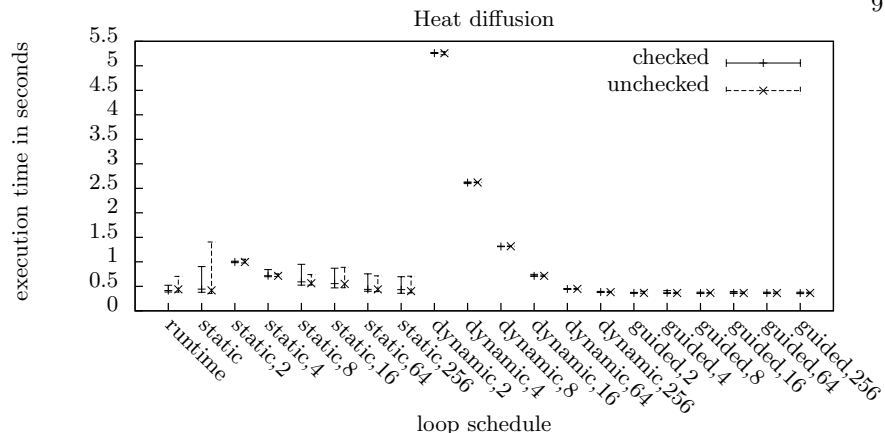


Fig. 4: Comparison of the maximal, minimal and average running times of *checked* and *unchecked* builds of the heat diffusion simulation with four threads and various selections of schedule and chunk size. The labels on the x-axis refers to the `schedule` clause.

The example uses a five-point stencil and thus the conservative estimate of inter-task dependencies among two executions of the stencil operation is five. Annotating the inner loop with a `taskshare` directive as shown in Listing 1.2 reduces that number to three between task pairs corresponding to a relative improvement of 40%.

The heat diffusion simulation was run for 2000 time steps with an array of 131072 doubles. The array sizes were set such that all data structures should fit in the CPU caches.

The running times of the heat diffusion simulation with and without runtime checks inserted are shown in Fig. 4. There were no statistically significant difference between the average running times of the binaries with and without runtime checks inserted.

The executions using the `dynamic` schedule and small values of the chunk size parameter have significantly higher execution times when compared to the other executions. This is because the `dynamic` schedule incurs a synchronization overhead each time a slice of iterations is mapped to a thread and the number of slices is in inverse proportion to the chunk size for the `dynamic` schedule.

Fig. 4 also shows a significantly higher worst-case execution time for uninstrumented builds for `runtime` and `static` schedules. Even when performing two or more warm-up runs before calculating average execution times, we saw that whichever build was executed first was also most likely to show a slightly higher worst-case execution time.

5.3 Demosaicing

An indispensable function in digital cameras is demosaicing which interpolates sensor data from a color filter mosaic. We used code developed for an embedded MPSoC [20]. It only uses stencils three elements wide which is too small to benefit from our `taskshare` directive but is nevertheless representative of a real

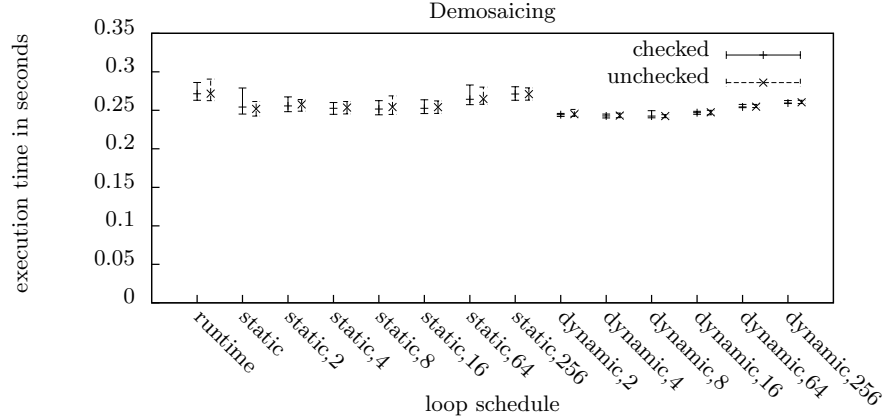


Fig. 5: Comparison of the maximal, minimal and average running times of *checked* and *unchecked* builds of the demosaicing program with four threads and various selections of schedule and chunk sizes.

world embedded application. Higher quality can be achieved with an algorithm using 5x5 stencils for interpolation [21]. A 21 mega-pixel image with a resolution of 5616x3744 pixels was used as input. The application contains three sequences of parallel stencil operations which were annotated.

No measurements were made with the `guided` schedule for this application since it triggers a known bug in the OpenMP runtime. All other results are shown in Fig. 5. There were no statistically significant differences between the average running times of the binaries with and without runtime checks inserted.

A slight increase in running time can be observed when increasing values of the chunk size parameter from 16 to 64 to 256 for the `static` schedule type. The effect also increases slightly when going from four to two threads. We are currently unable to give a reason for this but speculate that we are observing negative interference in the shared parts of the memory hierarchy, i.e., the L3 cache and memory controller.

5.4 Edge Detection

Detecting edges in an image is an important step in machine vision. We used the edge detection implementation which is part of the UTDSP benchmarking suite [22]. The input used consisted of 4096x4096 integer values.

The UTDSP edge detection code uses small stencils spanning only 3 elements which again does not require a `taskshare` directive. However, more advanced edge detection methods such as Canny’s edge detection use dynamically computed Gaussian stencils whose width typically range from 5 to 19 elements [23]. Lacking a more advanced edge detection implementation, we annotated the UTDSP edge detection benchmark. It contains a method with a single parallel stencil operation which is called multiple times with Gaussian and Sobel filters as input so a single `taskshare` directive was added.

The results are shown in Fig. 6. There were no statistically significant differences in the average execution time with and without checks inserted.

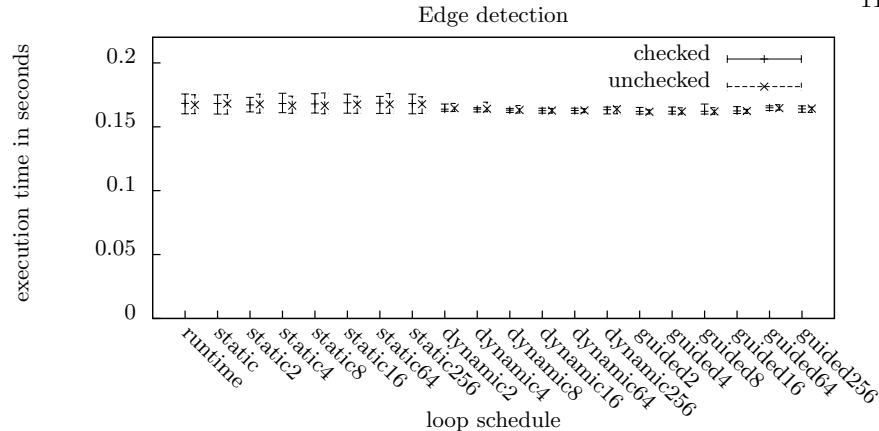


Fig. 6: Comparison of the maximal, minimal and average running times of *checked* and *unchecked* builds of the edge detection program with four threads and various schedules and chunk sizes.

6 Discussion and Conclusions

This paper presented a new directive to assist program analysis in computing inter-task dependencies. Specifically, it allows a reduction in the number of assumed dependencies in task graphs representing shared memory programs performing non-compact stencil operations. Such programs are found in embedded multimedia, machine vision, signal processing applications and in scientific computing.

Reducing the number of dependencies in task graphs i) can allow DSE tools to find more feasible designs and prevent over-provisioning of resources and ii) lower the running time of task scheduling algorithms whose asymptotic complexity increase with the number of dependencies.

When the stencil is compact, the inter-task dependencies can be correctly determined by program analysis. Whenever the stencil width is larger than three or unknown at compile time, the difference in the number of inter-task dependencies between the worst case and the common case grows with the number of threads.

The impact of the proposed directive was evaluated on the heat diffusion code, a single directive was added which results in a 40% decrease in the number of dependencies.

The embedded programs at our disposal used only 3x3 stencils and thus did not benefit from a `taskshare` directive. We argue that this is due to the simplicity of the particular implementations and can point to algorithms which use wider stencils to produce a higher quality output.

In none of the cases were there a statistically significant increase in the average running times due to the insertion of runtime checks. However, the low-overhead runtime checking, presented in this paper rests on the fact that stencil operations access data in a highly regular manner – less predictable codes will require more elaborate runtime checks.

Acknowledgements This work was partially supported by HiPEAC² and Artist-Design, both European Union Networks of Excellence. The demosaicing example was provided by Polytechnique Montreal and STMicroelectronics Ottawa. The research also made use of the University of Toronto DSP Benchmark Suite, UTDSP.

References

1. Schmitz, M.T., Al-Hashimi, B.M., Eles, P.: System-Level Design Techniques for Energy-Efficient Embedded Systems. Kluwer Academic Publishers (2004)
2. Mahadevan, S., Virk, K., Madsen, J.: ARTS: A SystemC-based framework for multiprocessor systems-on-chip modelling. *Des Autom Embed Syst* **11**(4) (2007)
3. Gries, M.: Methods for evaluating and covering the design space during early design development. *Integr. VLSI J.* **38**(2) (2004)
4. Sinnen, O.: Task Scheduling for Parallel Systems. Wiley-Interscience (May 2007)
5. Adve, V.S., Sakellariou, R.: Compiler synthesis of task graphs for parallel program performance prediction. In: Proceedings of LCPC '00
6. Cosnard, M., Loi, M.: Automatic task graph generation techniques. In: Proceedings of HICSS '95
7. Vallerio, K.S., Jha, N.K.: Task graph extraction for embedded system synthesis. In: Proceedings of VLSID'03
8. Liu, A.H., Dick, R.P.: Automatic run-time extraction of communication graphs from multithreaded applications. In: Proceedings of CODES+ISSS '06, ACM
9. Ahmad, I., Kwok, Y.K., Wu, M.Y., Shu, W.: Casch: A tool for computer-aided scheduling. *IEEE Concurrency* **8**(4) (2000)
10. OpenMP Architecture Review Board: OpenMP application program interface, version 3.0. Technical report (2008)
11. Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming. Addison-Wesley (2004)
12. Sarkar, V.: Partitioning and Scheduling Parallel Programs for Multiprocessors. MIT Press (1989)
13. Yang, T., Gerasoulis, A.: DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. Parallel Distrib. Syst.* **5**(9) (1994)
14. Ahmad, I., Kwok, Y.K.: On parallelizing the multiprocessor scheduling problem. *IEEE Trans. Parallel Distrib. Syst.* **10**(4) (1999)
15. El-Rewini, H., Lewis, T.G.: Scheduling parallel program tasks onto arbitrary target machines. *J. Parallel Distrib. Comput.* **9**(2) (1990)
16. Larsen, P., Karlsson, S., Madsen, J.: Identifying inter-task communication in shared memory programming models. In: Proceedings of IWOMP '09
17. Snir, M., et al.: MPI – The Complete Reference, Vol. 1, The MPI Core, 2nd ed. The MIT Press, Cambridge, MA, USA (1998)
18. Ha, S.: Model-based programming environment of embedded software for MPSoC. In: Proceedings of ASP-DAC '07
19. Bertels, K., et al.: HARTES toolchain early evaluation: Profiling, compilation and HDL generation. In: Proceedings of FPL '07
20. Bouchebaba, Y., et al.: MPSoC memory optimization for digital camera applications. In: Proceedings of DSD '07
21. Li, J.S.J., Randhawa, S.: High order extrapolation using taylor series for color filter array demosaicing. In: ICIAR '05, Springer
22. Lee, C.G., et al.: UTDSP benchmark suite. <http://www.eecg.toronto.edu/corinna/DSP/infrastructure/UTDSP.html> (1998) Date accessed: July 4th 2009.
23. Canny, J.: A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.* **8**(6) (November 1986)

A Comparison of some recent Task-based Parallel Programming Models

Artur Podobas¹, Mats Brorsson^{1,2}, and Karl-Filip Faxén²

¹ KTH Royal Institute of Technology

² Swedish Institute of Computer Science (SICS)

Abstract. The need for parallel programming models that are simple to use and at the same time efficient for current and future parallel platforms has led to recent attention to task-based models such as Cilk++, Intel TBB and the task concept in OpenMP version 3.0. The choice of model and implementation can have a major impact on the final performance and in order to understand some of the trade-offs we have made a quantitative study comparing four implementations of OpenMP (gcc, Intel icc, Sun studio and the research compiler Mercurium/nanos mcc), Cilk++ and Wool, a high-performance task-based library developed at SICS.

We use microbenchmarks to characterize costs for task-creation and stealing and the Barcelona OpenMP Tasks Suite for characterizing application performance. By far Wool and Cilk++ have the lowest overhead in both spawning and stealing tasks. This is reflected in application performance when many tasks with small granularity are spawned where Cilk++ and, in particular, has the highest performance. For coarse granularity applications, the OpenMP implementations have quite similar performance as the more light-weight Cilk++ and Wool except for one application where mcc is superior thanks to a superior task scheduler.

The OpenMP implementations are generally not yet ready for use when the task granularity becomes very small. There is no inherent reason for this, so we expect future implementations of OpenMP to focus on this issue.

1 Introduction

Now that parallelism is the only way forward to be able translate Moore's law into performance, it has become all the more important to find parallel programming models that are suitable for future manycore architectures from today's 4-64 cores to 100s in five years and 1000s in ten years and beyond [11,6].

We will soon have access to more cores than we will expect to utilize effectively at any given time. We may not even be able to run all at full speed for power reasons. Even so, scalability of application performance is of utmost importance to be able to deliver continued total system performance improvements. Some current approaches to parallel software development require the programmer to handle the complexity of performance scalability in addition to

exposing the parallelism in the underlying algorithms. We believe that this is a dead end to leverage the manycore technology at a wide scale. The vast majority of programmers must be able to focus on exposing potential parallelism with the aim for high-quality and high productivity. For portability and efficiency reasons, it should be left to the system software layers to deal with assigning work to available resources dynamically in run-time, although there is a need to expose it to programmers in special cases.

All of these considerations favour programming in high level programming models which provide *abundant fine-grained parallelism*, obviating the need for developers or compilers to explicitly map computations to the underlying hardware, which in any case is a moot approach in the face of dynamic workloads and heterogeneous hardware. This mapping is instead done by a run-time system that takes care of scheduling and resource management of the parallel activities. Such programming models are characterized by large numbers of dynamically created concurrent computations (tasks).

An important development are the task-parallel programming models such as exemplified by OpenMP [3,15], Cilk++ [5,13], and the Intel TBB framework [16]. This style is characterized by fine-grained parallelism that follows closely the structure of the application. For instance, a parallel loop can be implemented as a set of tasks corresponding to one or more loop iterations. The main form of synchronization is waiting for the completion of child tasks. Data parallelism can be realized as a higher level abstraction on top of task parallelism. Thus, efficient implementation of task parallelism also aids this model. A vital property of task-level parallelism is its ability to cope with heterogeneous and dynamically varying numbers of processing cores which is an inevitable result of future manycore development as we approach physical limits. While most implementations of this parallel programming model is done entirely in user-level libraries, there is at least one implementation where the model is integrated in the operating system [1].

This study is a performance comparison between six different implementations of task-parallel programming models. The models looked at are four implementations of OpenMP, Cilk++ and *Wool*, a new high-performance task library [9]. The four OpenMP implementations are: Gcc (v 4.4) [14], Intel Icc (v 11.0), Sun Studio 12 (update 1) and Mcc (Mercurium version 1.3.1 with Nanos run-time system version 4.1.3), a research compilation framework and run-time system from Barcelona Supercomputer Center [2,4]. For the study we have used a set of microbenchmarks developed by ourselves and applications from the Barcelona OpenMP Tasks Suite [8]. Although different schedulers and implementations have been compared before, this is, to the best of our knowledge the first to include *Wool* and to explicitly investigate the effects of fine-grained task-based parallelism [2,12].

We have found that the studied OpenMP implementations are not yet ready for fine-grained task parallelism. The associated overheads are by far too high. Cilk++ and *Wool*, on the other hand perform comparably well with a slight advantage for *Wool*.

2 Task-based parallel programming models

The task-parallel programming models represented by the implementations studied here were pioneered in the mid-90s by Blumofe et al. at MIT [5]. It was early recognized that the work-sharing constructs of OpenMP are not sufficient to express the potential parallelism in programs dominated by pointer-based data structures [17], but it took almost ten years to enter the OpenMP specification [3]. Wool was developed in order to further investigate the overheads associated with the task-based model and we have found that it indeed is possible to further push the limits.

Below is a short introduction to how each of the three models: OpenMP, Cilk++ and Wool implement task-based parallelism exemplified on a recursive calculation of the Fibonacci sequence.

2.1 OpenMP

OpenMP is a programming model that was created by a group that was representing several major vendors of high-performance computing [7]. It uses compiler directives and library routines to express and control parallelism. By adding these compiler directives to a sequential program, the users specify what parts are to be executed in parallel and how. As of version 3.0, OpenMP supports constructs for task-based parallelism while previous versions focused on loop based parallelism [15]. Figure 1 shows a code snippet of the fibonacci calculation. Parallelism is created by the “`#pragma omp parallel`” construct which creates a “team of threads”. The statement following the `single` directive is executed by the first encountering thread and kicks-off the recursive computation.

The compound statement following a “`#pragma omp task`” construct constitutes a computation which can be scheduled to be executed by any of the participating threads in a team of threads. A task may be executed immediately at the task creation or deferred to later execution by some other thread. By default, tasks are tied to the thread that starts executing it, so once a task has begun execution, it is then always executed by the same thread. The algorithm by which tasks are scheduled to available threads is not specified by the OpenMP specification but rather left to the implementation.

The `taskwait` construct suspends execution of the current task until all tasks created within this task has finished. In the example of Figure 1 this means that we are guaranteed to have valid values of variables `x` and `y`.

2.2 Cilk++

Cilk++ is model created and maintained by Cilk Arts, based on the original cilk model developed at MIT [13,5]. Through a small number of keywords, which are used to define possible parallel areas of a serial code, efficient parallel execution is realized. Removing these keywords from a cilk program creates a so-called “serial elision” of the program, which basically is a serial version of the programmed that can be used for debugging purposes. The Cilk++ scheduler is a work-first (also

```

...
#pragma omp parallel /* Parallel region, a team of threads is created */
#pragma omp single
    {
        /* Executed by the first thread */
        fib_result= fib(n);
    }
} /* End of parallel region */
...

int fib(int n) {
    int x, y;
    if (n < 2)
        return n;
    else {
        #pragma omp task shared(x)
            x = fib(n-1); /* A new task */
        #pragma omp task shared(y)
            y = fib(n-2); /* A new task */
        #pragma omp taskwait /* Wait for the two tasks above to complete */
            return x + y;
    }
}

```

Fig. 1. OpenMP Fibonacci

called depth-first) scheduler with a work-stealing mechanism where different idle workers can steal from other workers task pools. Figure 2 shows the example function written using Cilk++. `cilk_spawn` is the keyword for spawning a task, and `cilk_sync` will synchronize all spawned tasks with their parent.

The `cilk_spawn` and `cilk_sync` constructs are direct counterparts to the `task` and `taskwait` constructs of OpenMP. In contrast to OpenMP, the worker threads are completely implicit in Cilk++ and only the tasks are explicit. Also, the scheduling of tasks is predefined and not open for different implementations.

2.3 Wool

Wool is a library supporting the nested independent task parallel programming model [10]. It provides constructs for defining, spawning, and joining with tasks as well as for defining and invoking parallel `for` loops. Joining is accomplished using the `SYNC` operation which blocks until evaluation of the corresponding task is completed, providing for a direct, as opposed to continuation passing, program structure. Wool is designed to test the limits of low overhead task management. Figure 3 shows the example function written using Wool. `SPAWN` creates a task and `SYNC` synchronizes with the task, and fetches the return value off it. `CALL` is basically a faster version of a merged `SPAWN` and `SYNC`.

```

int fib(int n) {
    int x, y;
    if (n < 2)
        return n;
    else {
        x = cilk_spawn fib(n-1);
        y = cilk_spawn fib(n-2);
        cilk_sync;
        return x + y;
    }
}

```

Fig. 2. Cilk++ Fibonacci

```

TASK_1 (int , fib , int , n) {
    if (n < 2)
        return n;
    else {
        int x, y;
        SPAWN( fib , n-1 );
        y = CALL( fib , n-2 );
        x = SYNC( fib );
        return x + y;
    }
}

```

Fig. 3. Wool Fibonacci

Wool is implemented using work stealing, that is, each processor (core) has a private task dequeue; SPAWN pushes a task on to the *owner* end of the dequeue while a SYNC pops a task from the same end. When a processor is out of work it steals a task from the *thief* end of the task dequeue of a randomly selected *victim*.

The stacklike behaviour of SPAWN and SYNC is visible in the API since a SYNC always joins with the most recently spawned, unjoined task. SPAWN operations do not return a result; if the task is stolen, the result is stored in the task queue when ready and extracted by the corresponding SYNC. Tasks that are not stolen are invoked by the SYNC using the arguments stored in the dequeue, this is known as *inlining* the task. A SYNC operation takes as argument the name of the task definition of the task it expects to join with, so that the code to invoke when inlining the task is known and can be called directly, exposing it to compiler optimization.

Synchronization between thief and victim is based on individual task descriptors in the task dequeue rather than on the pointers into the queue, so that

the local processor's spawning and joining can take place independently of other processors looking for work.

Wool is implemented in C using macros, inline functions and a small amount of inline assembly (on x86 and x86-64 only to emit the `exchg` instruction). The SPARC V9, x86, x86-64 and IA64 architectures are currently supported. Wool is being developed at SICS.

3 Microbenchmark characterization

Most schedulers are built around a set of queues. The activities to be scheduled are stored in queues when not running on a processor. A simple scheme is to have a single queue to where all processors store work not presently running, but this scheme suffers from several performance problems. First, if work is fine grained relative to the number of procesors, there will be significant contention for access to the queue. Second, work tends to be distributed over the processors in a random fashion, while it would be advantageous to keep related work running on the same processor to maximize the effectiveness of caches. Hence most task schedulers use distributed queues, where each processor manages their own queue(s) with occasional coordination between processors for the purpose of load balancing.

In this section we present measurements of the basic operations of the different task schedulers using a small set of microbenchmarks. Specifically, we measure the overhead of using tasks as compared to running the same computation using procedure calls.

The use of distributed queues makes creation of parallelism a two stage process: First, new tasks are spawned to the local queue and later some (typically few) of the tasks move to execute on other processors. This enables very low overhead task creation since the first step typically does not entail communication between processors.

All experiments in this and the next sections were run on a Dell PowerEdge SC1435 dual quadcore Opteron server with 16GB of memory running Ubuntu Linux 9.04, kernel 2.28-15.

3.1 Experimental methodology

Measuring the cost of inlined tasks We have measured the cost of creating, spawning and joining with a task *on a single processor* by comparing the timings of the `fib` programs (given in figures 1, 2 and 3) when running on a single processor with that of a serial C program and dividing by the number of tasks created by the task parallel program. We have used different inputs for the different systems to arrive at execution times of a few seconds. This measures the marginal cost of a task over the cost of a procedure call in the case where the task end up being executed by the same processor that spawned it. Such tasks are referred to as *inlined* tasks in the work stealing context, and we extent the terminology to all of the systems investigated regardless of the scheduler used.

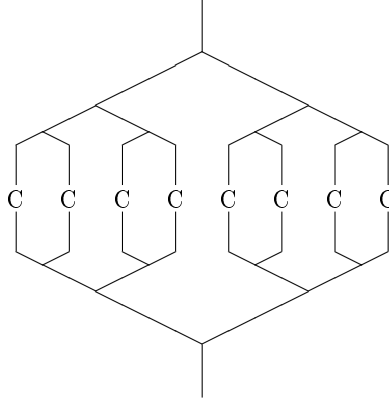


Fig. 4. The steal cost microbenchmark for 8 processors; C is a simple loop that makes no memory references

Inlining is the most common fate of a small task, especially in a work stealing implementation such as that of Cilk++ and Wool. The cost covers allocating, initializing and making the task available to the scheduler (spawning) as well as the cost of later joining with the not yet executed task (including the cost of synchronization and the cost of resuming its execution). We have attempted to disable optimizations that are unsafe when running on more than one processor, hence these timings include the cost of atomic instructions or memory barriers for the joining operation (sync or taskwait) although these are not strictly necessary for a single processor execution.

Measuring the cost of stolen tasks We measure the cost of stealing or migrating a task to a different processor (core) using a microbenchmark that repeatedly spawns and joins a balanced binary tree of tasks (see Figure 4), each of which executes a simple loop C making no memory references. The size of the tree is equal to the number p of processors used; consequently, the depth d of the tree is $\log_2 p$. We have measured the scaling behavior by varying d from 1 to 3 for two, four and eight processors, respectively. To compute the cost of spawning and joining the tasks in the tree, we compare the time to execute a depth d tree on 2^d processors (the parallel run) with the timing for a depth 0 tree (that is, just the loop C) on a single processor (the sequential run); the difference is the spawn/join cost. The number of iterations of the C loop is chosen individually for each system and value of d so that the difference in time becomes 10-20%, ensuring that we really get parallel execution of the tasks. Given that, the number of repetitions R is chosen to make the running time a few seconds. C and R are of course identical in the corresponding sequential sequential and parallel runs for a single system.

3.2 Performance results from microbenchmark study

As we can see from Table 1, Wool have significantly lower spawn/join cost than the other systems, and the consequences of that can be seen in the timings for the application programs. Since these measurements are from sequential runs,

Table 1. Costs of task creation/spawn/join on a single processor and across two, four and eight processors. All costs are measured in clock cycles.

System	Spawn/join (inlined)	Spawn/join (2 cores)	Spawn/join (4 cores)	Spawn/join (8 cores)
Wool	19	2 200	5 600	10 400
Cilk++	134	31 050	73 600	110 400
Gcc	415	5 200	16 800	52 800
Icc	878	4 830	9 200	20 240
Mcc	1 005	25 760	253 920	706 560
Sun cc	915	45 000	780 000	552 000 000

the overheads are only due to book keeping and not to effects like false sharing. This also explains why some systems are very dependent on limiting task depth.

When it comes to the parallel spawn/join benchmarks, the results show how well the synchronization and communication works. The ideal case here is low absolute time together with scaling with depth of the tree rather than the size (number of spawns) since the steals/migrations closer to the roots of the tree could in principle be performed in parallel. Of the tested systems, Cilk++ appears to be closest to scaling with the depth of the tree, although the costs are among the highest in absolute terms. Both mcc, gcc and the Sun cc compiler scales worse than linearly in the number of tasks spawned while Wool scales approximately linearly. The Intel compiler scales slightly better than linearly.

The Sun compiler stands apart in this comparison. Not only are the absolute costs higher than those of the other systems, but the scaling behaviour is excessive. Further studies are needed to understand this behaviour, but a preliminary analysis is as follows: We have observed that total CPU time in the eight processor case is less than eight times the elapsed time. Thus the processors have significant idle time. This phenomenon occurs when the number of available tasks is not much larger than the number of processors. For instance, running the benchmark with $d = 4 + \log_2 p$ brings the overhead to about five million cycles. Thus these results are probably related to the synchronization mechanisms used when accessing the task queue(s).

The microbenchmarks gives us some indications on the relative performance, but how is the effect on real application performance? This is investigated in the next section.

4 Application study

In this section we study and compare the performance of the six different models for five benchmarks from an early version of the Barcelona OpenMP task suite (BOTS) [8]. We ported the programs to Cilk++ and Wool, respectively, which was an easy task as these programs only used the functionality of the subset of OpenMP that is implemented in both Cilk++ and Wool. The five programs chosen are mostly taken from the original Cilk distribution (except SparseLU).

Table 2. The programs chosen to drive the performance comparison.

Name	Domain	Summary	Task size modifier
FFT	Spectral method	Calculates a Fast Fourier Transform	Size of vector before going serial. From 64k down to 16.
NQueens	Search	Finds solutions of the NQueens problem	Permitted task depth: 4, 8, 12 and 16.
Multisort	Integer sorting	Uses a mixture of sorting algorithms to sort a vector	Size of list before starting serial sort/merge. From 512k/256k down to 16/8.
SparseLU	Sparse linear algebra	Computes the LU factorization of a sparse matrix	N/A.
Strassen	Dense linear algebra	Computes a matrix multiply with Strassen's method	Size of sub-matrix before going serial. From 256 down to 16.

When this project started, we also had the programs Alignment and Floorplan from BOTS available but they either used OpenMP threadprivate variables or had other issues that made porting to Cilk++ and Wool not straightforward. Table 2 summarizes the five programs used. The default workloads of each program as implemented in BOTS have been used.

In the next sections we show the relative performance of the six different task-based models/implementations. The programs are configured so that the task-depth in recursive calls can be controlled. SparseLU does not have recursive generation of tasks, so this program naturally, does not have this. Table 3 shows the compiler flags used in each case.

Table 3. Compiler flags used in the experiments.

Compiler	Flags
Serial (gcc)	-O3 -m64 -static
Wool	-O3 -m64 -lpthread -lm
Cilk++	-O3 -m64 -static
Gcc	-O3 -m64 -fopenmp -static
Icc	-O3 -m64 -openmp -openmp-link static
Mcc	-O3 -m32 -v -k
Sun cc	-O3 -m64 -xopenmp=parallel

Most of these programs, except SparseLU, are recursive in nature and one of the main objectives of this study has been to investigate as to how OpenMP and the other models fare when we allow deep recursions as it is either difficult to make automatically in the run-time system or cumbersome for programmers to do themselves. This is modified for the different programs according to the task size modifier specified in Table 2.

4.1 Performance results

Parallelism overhead The first experiment reports on the overhead created by the parallel constructs in Wool, Cilk++ and OpenMP, respectively in sequential program compared to the parallel programs with one thread. Table 4 shows the result of this measurement normalized to the sequential execution of each program. In these experiments, as in all others in this section, the programs have been executed ten times on an otherwise unloaded machine and the mean execution time taken.

Table 4. Overhead of parallel constructs for the five programs and six models.

Compiler	FFT	NQueens	Multisort	SparseLU	Strassen
Wool	0.96	0.98	0.96	0.97	0.76
Cilk++	0.93	0.78	0.93	0.96	0.76
Gcc	0.95	0.91	0.97	0.88	0.72
Icc	0.98	0.93	0.94	0.98	0.84
Mcc	0.91	0.95	0.98	0.99	0.55
Sun cc	0.73	0.89	0.92	0.99	0.94

The results are somewhat inconclusive. None of the compilers is consistently best, although Icc seems to have a robust performance in this respect. All others have really poor performance for at least some benchmark. Future studies will investigate the sources of this in more details.

Overall performance Figures 5 to 9 show the speedups relative the serial execution for all compilers using 1-8 processors. To the left in each figure is shown the default coarse grain task granularity where a cutoff in the generation of recursive tasks is set quite early. To the right (except for SparseLU) is shown a fine-grained generation of tasks for which we discuss the results in section 4.1.

When task-granularity is coarse, all compilers perform relatively well. The difference in performance can be attributed more to difference in compiler optimizations rather than in the implementation of parallelism. For Multisort, Mcc clearly outperforms all other compilers and the full reason for this remains to be investigated. Profilers show that the programs spend most of their time in the run-time system for multisort so the implementation of task scheduling is crucial here. Sun CC outperforms all other compilers for the Strassen benchmark. The reason here is superior code quality and in particular much lower branch miss prediction penalty as indicated by experiments using AMD CodeAnalyst.

Importance of task-depth cut-off The real interesting results are shown when we allow the program to generate many fine-grained tasks. In all cases, the OpenMP compilers really perform poorly when the task granularity becomes small. One of the main advantages of using a task-based model is that the user

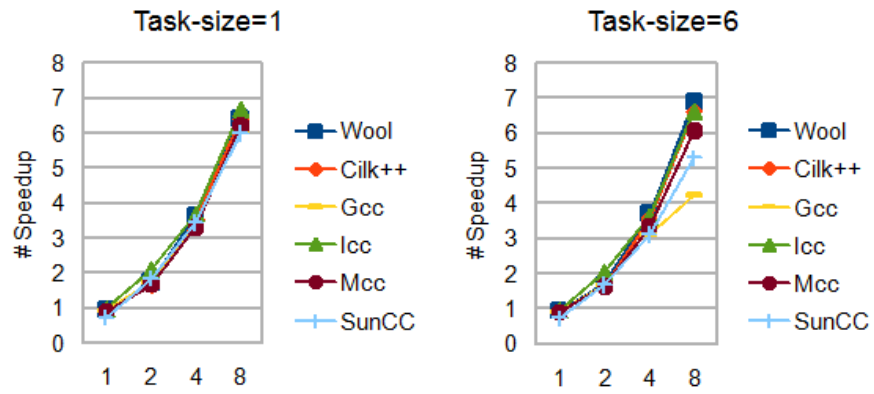


Fig. 5. FFT speedup with coarse and fine grained task granularity.

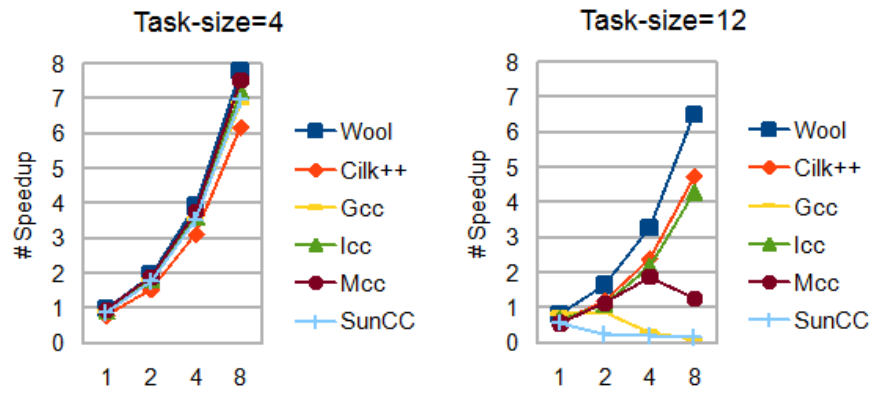


Fig. 6. NQueens speedup with coarse and fine grained task granularity.

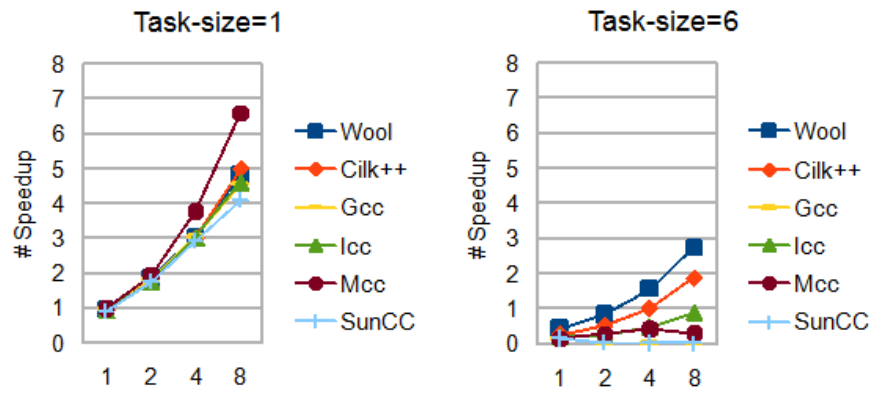


Fig. 7. Multisort speedup with coarse and fine grained task granularity.

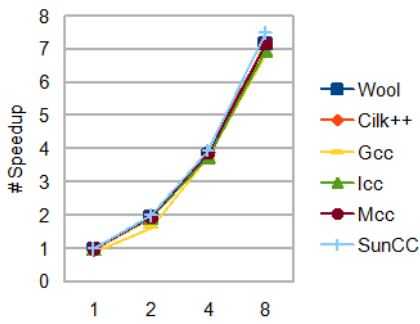


Fig. 8. Sparse LU Speedup.

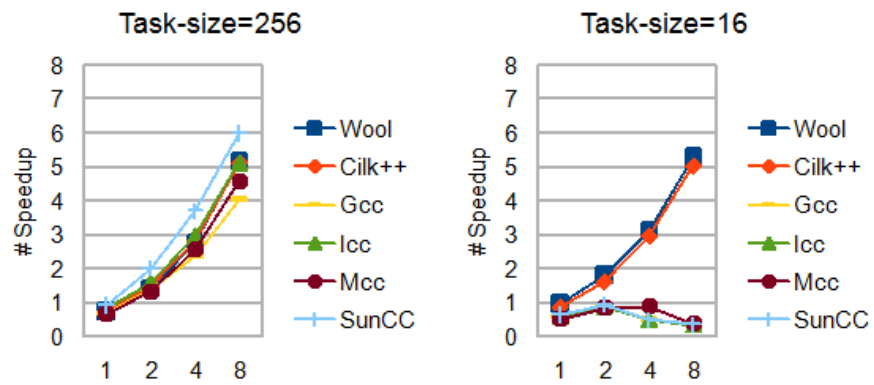


Fig. 9. Strassen speedup with coarse and fine grained task granularity.

should be able to concentrate on exposing the parallelism inherent in the application rather than trying to figure out how it would fit on this particular machine.

Furthermore Wool consistently outperforms Cilk++ for all fine-grained applications. In some cases significantly. We have paid great detail in specifically making the spawn process cheap and also to provide for a low-overhead and scalable work-stealing algorithm which pays off for programs such as NQueens and Multisort.

5 Conclusions

We have reported on performance results for a number of task-parallel programs for six different implementations of task-based parallel programming models. It is clear that the OpenMP implementations studied work well for coarse grained applications but equally clear that they fail miserably for fine-grained tasks. One of the main advantages of task-based parallelism is that the programmer can concentrate on exposing available parallelism instead of worrying on scheduling the computations on threads manually which is what you have to do with normal thread-centric models as pthreads and also to some extent with OpenMP ≤ 2.5 . There is no inherent reason that OpenMP should perform this badly for fine-grained applications so we expect that this will be an area of focus for future implementations.

This study is, to the best of our knowledge, the first to compare Wool with the other major models and also to look at finer granularity of tasks. We are happy to note that the comparison puts Wool into a good position, but there are also a number of questions that we need to further study in order to understand what is happening. Why is Sun cc so incredibly bad in the micr-benchmark but reasonably effective for the coarse grained applications? Why is mcc considerably better for the multisort application? Why are the parallelism overheads varying so much for the different programs? All of these questions as well as scalability issues for larger core counts will be the focus of future studies.

References

1. Grand central dispatch. Technology Brief, 2009. Retrieved Sept 29, 2009 from <http://www.apple.com/macosx/technology/>.
2. E. Ayguadé, A. Duran, J. Hoeflinger, F. Massaioli, and X. Teruel. An experimental evaluation of the new openmp tasking model. *Lecture Notes In Computer Science*, 5234:63–77, 2008.
3. Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.
4. J. Balart, A. Duran, M. González, X. Martorell, E. Ayguadé, and J. Labarta. Nanos mercurium: a research compiler for openmp. In *Proceedings of the European Workshop on OpenMP*, volume 2004, 2004.

5. R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216. ACM New York, NY, USA, 1995.
6. S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual conference on Design automation*, pages 746–749. ACM New York, NY, USA, 2007.
7. L. Dagum, R. Menon, and S.G. Inc. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science & Engineering*, 5(1):46–55, 1998.
8. Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguadé. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *38th International Conference on Parallel Processing (ICPP '09)*, page 124–131, Vienna, Austria, September 2009. IEEE Computer Society, IEEE Computer Society.
9. Karl-Filip Faxén. Wool-a work stealing library. *SIGARCH Comput. Archit. News*, 36(5):93–100, 2008.
10. Karl-Filip Faxén. Wool user’s guide. Technical report, Swedish Institute of Computer Science, 2009.
11. International technology roadmap for semiconductors. <http://www.itrs.net>, 2007. Retrieved on Sept 28, 2009.
12. Olivier Stephen L. and Prins Jan F. Evaluating OpenMP 3.0 Run Time Systems on Unbalanced Task Graphs. In *Fifth International Workshop on OpenMP*, June 2009.
13. C.E. Leiserson. The Cilk++ concurrency platform. In *46th Design Automation Conference, San Francisco, CA*, 2009.
14. D. Novillo. OpenMP and automatic parallelization in GCC. In *GCC developers summit*, 2006.
15. Openmp v. 3.0 specification. <http://www.openmp.org>, 2008. Retrieved on Sept 28, 2009.
16. J. Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O’Reilly Media, Inc., 2007.
17. S. Shah, G. Haab, P. Petersen, and J. Throop. Flexible Control Structures for Parallel C/C++. In *First European Workshop on OpenMP, September*, September 1999.

Handling of shared memory in many-core processors without locks and transactional memory

Andras Vajda¹,

¹ Ericsson SW Research,
Hirsalantie 11,
02420 Jorvas, Finland
{Andras.Vajda}@Ericsson.com

Abstract. Handling of shared data structures efficiently and in a scalable manner in many-core processors is one of the key items on the research agenda. There are two basic solutions proposed so far, lock based and transactional memory based solutions, respectively. However, both of these have various limitations related to composability and scalability in general and predictability in special when applied to systems with e.g. real-time constraints. In this paper we propose a novel approach that addresses these limitations and provides a way to converge on a predictable, composable and simple approach – from the programmer’s perspective – for handling shared state in many-core processors. It builds on the possibility of mitigating problems related to concurrent accesses by strictly enforcing data locality and dedicating some of the processor cores (of which there may be several hundreds in future chips) to specific tasks. This approach guarantees earliest possible access to the shared data areas for each entity (thread or processor core) requesting it and dynamically converges to a state with no deadlock or abort situations, while automatically detecting in real-time and resolving deadlocks that may occur.

Keywords: Many-core processors, shared memory, locks, transactional memory.

1 Model Description

There has been a significant amount of work done recently addressing the problem of shared data structures in multi-processor systems, especially in chip multiprocessors (CMP). There are fundamentally three approaches to mitigate the problem of access to shared state:

- Using locks

- Using HW/SW transactional memory
- Lock-free data structures

All three of these approaches have a number of limitations:

- Locks are *non-composable*, i.e., two pieces of correct program code, when combined, may not perform correctly, leading to hard-to-detect deadlock or live-lock situations
- Transactional memory solutions, while composable, have a *significant processing overhead*, usually require HW support and software realizations *do not scale well*: the system will perform increasingly inefficiently in case the number of processing elements trying to access the same data is increased; it has been shown previously that e.g. Haskell's STM solution does not scale beyond 8 cores in a CMP ([11])
- Neither locks nor STM solutions are *predictable and deterministic*, i.e., it's very difficult – and in some cases impossible – to calculate a reliable upper-bound for execution time; this behavior is not suitable for e.g. real-time applications
- Lock-free data structures and algorithms require case-by-case development and there is no universally applicable solution available.

To mitigate these problems, we propose a solution that will address these issues, leveraging on new possibilities provided by many-core processors. The proposed solution leads to an improvement in the utilization of the memory bandwidth to the processor, hence enhancing further the usability of the solution.

1.1 Basic Assumptions and Principles

We build our proposal starting from the following three basic observations and assumptions:

1. The best way to avoid shared memory access conflicts is to enforce that there is only one and the same processing entity that has access to the shared memory
2. In future many-core processors, with several hundreds of cores, memory bandwidth and on-chip cache capacity will be the bottlenecks, while sheer processing power is a plentiful resource where limited waste is acceptable
3. Usually the size of code is smaller than the amount of data it works on, hence keeping data on chip and moving code between the chip and off-chip memory shall reduce the memory bandwidth requirements

Based on these three assumptions, we propose a system where access to shared memory is always done by one and the same processing core which will perform the requested operations according to a configurable policy, on behalf of threads needing access to that memory location. This way, a strict serialization of the accesses to the shared resource is obtained and it is guaranteed that there are no race conditions. In addition, according to point 3) above, memory stalls due to data missing from on-chip memory can be reduced by keeping the shared memory areas in the cache of the only core that may access it.

The proposal is structured into two major parts: the application level model and the run-time system that implements the principles above.

1.2 Programming Model

On the programming model level, there are two principles that need to be adhered:

1. Shared data areas shall have a unique id, that we will subsequently call *shared data id (SDI)*
2. All pieces of code accessing shared data shall be marked as *critical sections (CS)* along with an explicit listing of the SDIs of all shared data areas accessed by the critical section

Principle 1 does not explicitly require that shared data areas and SDIs are matched to each other; in fact, SDIs have the primary goal of identifying which shared data areas are accessed together, by the same critical section. However, supporting explicit matching of actual memory locations and shared data ids can help optimize data placement by the compiler, as we will describe in the run-time system section.

Principle 2 only requires that the programmer clearly delimits the sections in the code that will have access to shared data areas and which those shared data areas are. There is no need to add any synchronization primitives or any detailed analysis of race conditions, as these will be taken care of by the run-time system. A simplification of critical sections – with positive impact on memory performance – is to define these as (almost) pure functions, in the sense that the critical section will only access critical section (function)-local data and shared data areas that it is explicitly listing, but no other global or thread-local data. This way, as access to thread local data is not needed, cache misses can be avoided. We will call such critical sections *pure critical sections*.

One possible enhancement of critical sections is to allow critical sections to be either blocking or non-blocking. A blocking critical section will have to finish first before the thread from which it is invoked can proceed; a non-blocking critical section on the other hand allows the continuation of execution of the invoking thread. In this case, if the invoking thread needs the results of the critical section (e.g. access the same shared data areas), a synchronization mechanism needs to be introduced, as we'll show next.

In its most generic form, the programming model will require the following two mandatory constructs and a third optional one:

1. Declaration of global shared memory areas:

SHARED (SDI)

```
{ // data definition};
```

2. Definition of critical sections:

CRITICAL <CSNAME> SHARED SDI1, SDI2, ... SDIn [NON-BLOCKING]

```
{ // actual code of the critical section};
```

Semantics: defines a sequence of code as a critical section, accessing global shared memory locations identified by *SDI1, SDI2, ... SDIn*. If the indicator **NON-BLOCKING** is present, the critical section's execution will not suspend the current

thread; rather the two will execute in parallel. This feature is useful in case of updates to e.g. a global counter.

3. (Optional) Synchronization with (one or several) non-blocking critical sections:

SYNCH (<CSNAME1>, <CSNAME2>, ...<CSNAME3>)

Semantics: the current thread will wait until critical sections with names CSNAME1, CSNAME2, ... CSNAME_n have been executed. Note, the critical sections shall be marked as **NON-BLOCKING**.

1.3 Run-time System

Based on the three basic principles at the foundation of our proposal, each shared data area, identified by its unique SDI, shall be accessed by one and always the same processing element (processor core in a many-core chip). Also, assuming limited waste of processing cores acceptable, we will dedicate some cores for executing code accessing shared data areas as a primary task. Hence, on the HW level cores will have two primary roles:

- *User Processing Entity (UPE):* a processing entity executing normal application code
- *Resource Guardian (RG):* a processing entity that has the exclusive access to a shared data area, executing critical sections based on requests received from UPE instances

Clearly, it's not a workable approach to allocate a different RG for each shared data area. Instead, we define the concept of *critical section group (CS-G)*, as a group of CS with the property that any CS in the group accesses at least one shared data area that is accessed by at least one another CS in the group; basically this means that critical sections in a group are depending on a common set of shared data areas (and hence may have an execution order dependency on each other). Each CS-G will have one RG allocated to it, that will 'own' all the shared data areas that the CS in the group are accessing. *Figure 1* gives examples of CS-G.

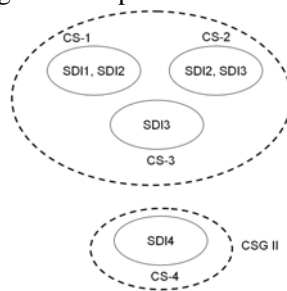


Figure 1 Examples of CS Groups

In concrete implementations, a core executing as RG may also execute other applications as well. In this case however, the cache of the core may be polluted with other data and hence the efficiency of access to shared data areas will suffer.

1.3.1 Execution on UPE

UPEs execute user applications normally until the start of a critical section is reached. At this point, the UPE shall generate a message sent to the task queue (see definition in the next section) of the RG dedicated to the CS-G to which the current critical section belongs; after this task is generated and dispatched the UPE has one of the two following choices:

- If the critical section is marked as blocking, the UPE will execute a SYNCH, waiting for the critical section's execution to complete on the RG(s)
- If the critical section is marked as non-blocking, the UPE will continue the execution of the program right after the critical section

When the UPE encounters a SYNCH construct, it shall halt the execution of the current thread until a reply is received for all critical section invocations listed in the construct from the corresponding RG(s).

1.3.2 Execution on RG

In order to guarantee that no deadlock will occur, there shall be exactly one RG for each critical section group, and there shall be a task queue per priority level for this RG. A *task queue* is a queue with requests (tasks) to execute critical sections; each task shall include identification of the initiating thread and processing element, location of the critical section that needs to be executed as well as a list of RGs that have already been 'visited' by this task (relevant in case of nested critical sections, see in subsequent sections). There may be multiple task queues per RG, one for each priority level. *Figure 2* gives a graphical overview of the concepts.

The logic on the RG is as follows:

- Fetch the next task from the queues by the rule:
The task shall be the highest priority of all the executable tasks

If no such task is found, the RG will idle until a task with these characteristics will be available

- Once a task is selected the corresponding critical section is executed
- Once the execution of the critical section is completed, a completion message is sent to the invoking UPE indicating successful completion of the task and the next task will be selected.

This solution allows for multiple optimization opportunities. First, the shared data areas may be kept, as much as possible, in the core-local memory (or cache); in fact, a dedicated hardware architecture may have bigger local caches for cores that will be allocated the role of resource guardians. The existence of task queues with deterministic ordering for tasks also provides a good basis for predictive pre-fetching of e.g. critical section code from main memory.

One of the key issues to be addressed is the handling of *nested critical sections*. In the following section we propose an approach which will allow for composability and automatic resolution of dead-lock situations, without need for programmer intervention.

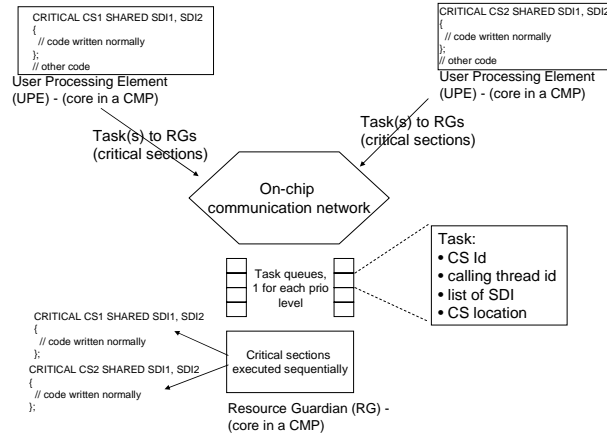


Figure 2 Concepts

1.3.3 Nested Critical Sections

It is possible that a critical section invokes another critical section. If the nested critical section is member of the same CS-G, the nested critical section will be executed immediately; in other cases a more sophisticated approach is needed.

A potential source for deadlock occurs if the nested critical section belongs to a different CS-G. In this case, the RG itself will have to generate a new task to the RG owning the shared data area(s) accessed by the nested critical section; however, several cases may lead to deadlocks. For example, if there are two CS-Gs – A and B – the following scenario may potentially lead to a deadlock situation:

- RG corresponding to CS-G A (“RG A”) is executing a critical section; at the same time, the RG corresponding to CS-G B (“RG B”) is executing an other critical section
- The critical section executed by RG A is invoking a critical section belonging to CS-G B, which will lead to RG A suspending execution until the invoked critical section is executed on RG B
- If a same occurs on RG B, invoking a critical section belonging to CS-G A, there is a deadlock: RG A & B will be mutually waiting on each other

We propose a mechanism that allows resolution of such cases.

The key for automatic resolution of possible dead-lock situations lies in co-operation between RGs. Firstly, whenever a nested critical section is invoked, the task that will be generated shall include the list of RGs that are already blocked by the current nested chain of RGs. This information will be used to handle a specific sub-case of deadlocks, as will be explained below.

Secondly, we propose that each RG will inform all other RGs when it will go to a blocking mode, including also the reason, i.e., which RG it is invoking. Similarly, when it’s de-blocked, it will inform the other RGs as well. These messages can be

processed by the RGs even if these are in blocked state, i.e., waiting for another RG to complete and may be handled as prioritized tasks that will always get the highest priority. The amount of additional activity – in terms of number of tasks and on-chip traffic – will be fairly low, as these special tasks will only be generated in case of nested critical sections and the tasks themselves will only be a few bytes (or tens of bytes) long.

This mechanism allows each RG to build a *RG dependency graph*, showing which RG is at any given moment blocked and what is the reason (i.e., which RG it is waiting for). The RG dependency graph is built with the following semantics:

- The nodes are the RGs which are blocked or are executing requests from a blocked RG
- The directed links (from A to B) indicate that the RG corresponding to node A is waiting on the RG corresponding to node B to complete the execution of a nested critical section

The RG dependency graph has some interesting characteristics:

- It only contains the RGs that are currently blocked or are executing nested critical sections invoked by blocked RGs
- Each node may have any number of predecessors but exactly one successor – the RG it is waiting for

Each RG will build a copy of the RG dependency graph based on the messages received from other RGs that go blocked state or resume execution. The RG dependency graph has a key property: *it is loop free if and only if there is no dead-lock*; any loop in the graph indicates a deadlock, involving exactly the RGs of the loop. The proof of this is straightforward, considering the rules for building RG dependency graphs described above. See *Figure* for an example.

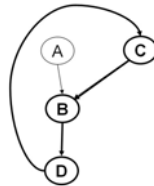


Figure 3 Example of RG dependency graph with dead-lock

Using this property, any RG can reliably and deterministically detect any deadlock situation; however, breaking the deadlock situation shall be initiated by the RGs involved in it. This has the added advantage that the blocked RGs cannot perform any other useful work anyway, thus identifying deadlocks (loops in the RG dependency graph) shall not take away computing resources from other tasks.

The first measure taken relies on the fact that each task includes the RGs that are already involved in the execution of the respective chain of nested critical sections. If a blocked RG receives a task indicating it as reserved, it may execute the task immediately, as it is obvious that it got blocked for the chain of critical sections to which the task belongs; in this case the dead-lock is in fact just a circular, recursive access to the same resources and execution can safely continue, as long as the RGs are re-entrant (can accept new tasks while waiting for other RGs to complete).

In case a real deadlock is detected (there's a loop in the RG dependency graph that does not fall under the previous paragraph), there shall be a policy that allows the selection of an RG that will break the deadlock. The only possible way to break a deadlock is to roll back the execution of one of the critical sections that led to the deadlock; hence the selection of RG implies effectively the selection of one of the critical sections. Possible policies include:

- RG blocked with the shortest list of depending (already blocked) RGs – in order to minimize potential amount of rollbacks that need to be performed
- RG blocked by the critical section with the lowest priority
- Usage of rankings of RGs and roll-back by the RG with the lowest rank

It's important to note that all RGs on the deadlock chain will immediately detect the deadlock; hence selection of an RG is a matter of prioritization between the RGs on the deadlock chain.

Once a RG is selected, it shall perform the following actions:

- 1 Roll back the critical section that led to the blocking state
- 2 Inform the RG where the nested critical section was dispatched that the task shall not be executed anymore
- 3 Inform other RGs that it's not dependent on the respective RG anymore
- 4 Execute the task received from the RG preceding it in the RG dependency graph and belonging to the deadlock chain
- 5 Resume normal execution

Step 1 can be implemented using a simple shared data area mirroring system. This can be realized by always keeping two copies of the memory area containing the shared data owned by the RG (active and backup) and simply switching to the backup version. This may be achieved by memory copying or by HW support.

1.3.4 Automatic Dead-lock Avoidance

One key feature of the method described in this paper is the possibility of improving system behavior based on collecting information on dependencies between RGs. For example, detecting that two RGs from different CS-Gs are actually dependent on each other, or – even worse – are responsible for a deadlock can be used to merge the two CS-Gs in order to reduce dependencies and avoid execution hand-over between RG instances. Especially in the case of detection of deadlocks, merging all CS instances participating in the deadlock into one CS-G will lead to avoidance of deadlock in the future. Such a feature will guarantee that the system will converge to a stable, deadlock-free state autonomously, with no external intervention.

The merge of RGs shall however be subject to a carefully chosen strategy: if a deadlock only occurs rarely, automatic resolution may still be preferable to merging RGs in order to preserve parallelism; however a systematically recurring deadlock pattern may be a prime candidate for an RG merging decision. The key benefit of the method described in this paper is that it allows elastic scoping of RGs, fast and reliable detection of deadlocks, a simple scheme for resolving deadlocks as well as a mechanism to avoid future deadlocks following the same pattern as already detected deadlocks.

2 Discussion

It can be proven by a simple logical deduction that this approach guarantees the earliest possible execution of any critical section, assuming that all critical sections within a critical section group are inter-dependent and taking into account the priority levels. Indeed, looking at the possible scenarios, it can be seen that, at the moment a task for a critical section is fired, either

- The corresponding RG task queue is empty and the task will be handled immediately, or
- The corresponding RG task queue already contains other tasks that were issued before this task and hence shall be executed first; however, this task will be executed as soon as all other previously issued tasks are handled, i.e., at the earliest possible moment.

There are however cases when two critical sections belonging to the same critical section group could execute in parallel. For example, assuming three critical sections CS1 (SDI1, SDI2), CS2 (SDI2, SDI3) and CS3 (SDI3), these will belong to the same CS-G; CS1 and CS3 could nevertheless execute in parallel. However, allowing parallel execution of CS1 and CS3 may lead to deadlock situations, in case e.g. CS1 contains a nested critical section requiring access to SDI3 and CS3 containing a nested critical section requiring access to SDI1 & SDI2. One possible approach could be to allow such parallel execution – by e.g. having multiple RGs for each CS-G – and use STM between the RGs. Alternatively, the CS-G shall be made more fine-grained by splitting up the critical sections into smaller entities.

Based on the discussion above, the solution is guaranteed to compose, by virtue of enforcing strict locality of data and guaranteeing that there is always one and only one core accessing a certain shared memory area. In addition, as deadlocks are automatically resolved and eventually eliminated, the system can tolerate and adjust to any composition of software with any pattern of nesting critical sections.

Also, by virtue of the same characteristic – enforcing locality of data – and through the detection and merging mechanisms for the CS-G, the system will converge to a stable state where a critical section will never fail, simply because it's guaranteed that it is the only one piece of code accessing a certain shared access resource. This characteristic suits well the requirements of real-time systems which put high focus on predictability.

The method described herein can also be used for fine-tuning a system: allowing a system to run for a certain period of time and examining the encountered dead-lock situations as well the CS-G merging decisions taken by the run-time system, the code itself can be optimized to avoid dead-locks that were not apparent initially (but were detected and solved by our method).

Enforcing data locality and code locality for data manipulation offers the chance of keeping both the data and the critical section code local to the RG (in the L1 or L2 cache closest to the RG), by this enhancing the performance of the system by avoiding cache misses, by avoiding costly memory accesses unless absolutely necessary and by avoiding the need for invalidating cache lines and re-fetching the most up-date values.

2.1 Empirical Results

The most performance critical aspects of our proposal that we wanted to verify experimentally was the performance of task firing versus obtaining a lock/usage of transactional memory, and cache behavior. Based on prototype implementations under Linux, we found that task firing / consumption consumes equal or up to 20% less cycles on a 64 core machine than obtaining a lock in non-contested context (no other application held the lock). It's important to note that we relied on tasks being sent over the on-chip network rather than using shared memory. Based on previous work, comparing locking and initiation of a transaction in a software transactional memory (STM) environment, same or better performance would be obtained in comparison with using STM.

When it comes to memory usage efficiency and cache performance, our method has shown a dramatic improvement compared to traditional systems. The gain varied according to the size of the shared data area, the size of the cache and the size of critical sections, but it was, in an experimental setup, always at least 50%, sometimes up to 200% (in case of small caches and data areas larger than the cache size).

Complete implementation of the method described herein is still work in progress; hence these empirical results show the partial picture. However, we expect that, by seeing comparable or better performance for the most critical aspects of the proposal, the final results to show a marked improvement in performance, while off-loading the burden of managing complexity of access to shared data by the programmer.

3 Conclusion

In this paper we presented a novel approach for handling shared resources in a many-core system. The proposal builds on the availability of enough parallel computing resources (i.e. processor cores) to allow off-loading access to shared resources to single points of access through which locality can be enforced. In addition, we describe a method for automatically detecting and resolving deadlocks as well as for dynamically adapting the run-time configuration of the system so as deadlocks following similar patterns can be avoided.

The solution is deterministic and composable, under the assumption that the shared resources and the critical code sections addressing these can be grouped in disjunctive sets, each handled by different resource owning processing elements (cores). The solution also improves memory performance by allowing efficient caching of shared memory resources and code that accesses these resources and thus avoiding costly cache misses. The proposed solution may be combined with the methods described in [14], [16] in order to achieve better memory performance by pre-fetching memory content to local cache, based on the inspection of the task queue on the RG cores.

Our proposal also contains an easy to implement mechanism for the detection of deadlocks as well as a solution for dynamically adapting the system in order to avoid future deadlock situations. This way, the system converges to a deadlock-free, stable and predictable state, suitable for real-time, periodic systems. The key feature here is

the elastic scoping of the resource guardians: the run-time system can dynamically adapt the RGs so that the scope matches the actual application characteristics.

Future work shall concentrate on further enhancing parallelism, such as allowing multiple RG instances for each critical section group. Prototype implementation in order to evaluate how such solution will integrate with operating systems and existing run-time systems shall also be completed.

4 Related work

The advantages and disadvantages of transactional memory and locking-based schemes have been extensively researched and documented in recent years. The fundamental research results on transactional memory are available in [1], [2]. Strengths of and problems related to locking have been extensively explored in [3], [4], [5], [6], [8], while problems and improvements of transactional memory solutions are analyzed in [1], [7], [9], [10], [11]. Paper [12] provides a good comparison of locking and transactional memory strategies and possibilities for improving these.

Paper [13] introduces the concept of threads flowing from one core to another, an idea which can be found also in our proposal, through the inheritance of thread context by the resource guardian cores. Paper **Error! Reference source not found.** proposes an approach similar to ours, however without the flexibility of multiple RGs, automatic dead-lock resolution or scoping of RGs, concentrating all the shared resource accesses to one (fat) core.

The concept of using another core to perform some tasks on behalf of the ‘main’ execution core has been exploited in [14] where it was used to perform memory pre-fetching based on memory usage patterns. In [16] a method for detecting memory access patterns and implementing HW logic to help memory pre-fetching is described; these methods can be used to improve RG cache performance, as both the code and data access patterns are known exactly (through ownership of SARs and inspection of task queues).

As part of the SERVO programming model [15], a proposal was made to handle shared data structures as services, managed by a dedicated software instance. That approach, however, stopped short of allowing execution of arbitrary code, accessing an arbitrary number of data services and besides, the declaration of the data services would need to be explicit and access code for the shared data would need to be explicitly written, using the run-time library. In addition, this approach does not support execution of the access in the context of the main application thread.

References

- [1] Herlihy, M. and Moss, J.E.B. Transactional Memory: Architectural support for lock-free data structures. *Annual International Symposium on Computer Architecture*, (May 1993), 289 – 300.

- [2] Herlihy, M. The transactional manifesto: software engineering and non-blocking synchronization. *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation* (2005), 280.
- [3] Hoare, C.A.R. Monitors: An operating system structuring concept. *Communications of the ACM* 17 (October 1974), 549 – 557.
- [4] Inman, J. Implementing loosely coupled functions on tightly coupled engines. *In USENIX Conference Proceedings* (Portland, OR, June 1985), USENIX Association, 277–298.
- [5] Kontothanassis, L., Wisniewski, R. W., and Scott, M. L. Scheduler-conscious synchronization. *Communications of the ACM* 15, (January 1997), 3–40.
- [6] Lampson, B. W., and Redell, D. D. Experience with processes and monitors in Mesa. *Communications of the ACM* 23, 2 (1980), 105–117.
- [7] Marathe, V. J., Spear, M. F., Heriot, C., Acharya, A., Eisenstat, D., Scherer III, W. N., and Scott, M. L. Lowering the overhead of nonblocking software transactional memory. *In TRANSACT: the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing* (June 2006), ACM SIGPLAN.
- [8] McKenney, P. E. Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels. *PhD thesis*, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.
- [9] Moore, K. E., Bobba, J., Moravan, M. J., Hill, M. D., and Wood, D. A. LogTM: Log-based transactional memory. *In Proceedings of the 12th Annual International Symposium on High Performance Computer Architecture (HPCA-12)*, (Washington, DC, USA, 2006), IEEE.
- [10] Scherer III, W. N., and Scott, M. L. Advanced contention management for dynamic software transactional memory. *In Proceedings of the 24th Annual ACM SIGOPS Symposium on Principles of Distributed Computing*. Association for Computing Machinery, (July 2005), 240–248.
- [11] Perfumo, C., Sönmez, N., Stipic, S., Unsal, O., Cristal, A., Harris, T. and Valero, M. The limits of software transactional memory (STM): dissecting Haskell STM applications on a many-core environment. *Proceedings of the 2008 conference on Computing frontiers* (2008), 67 – 78.
- [12] McKenney, P.E., Michael, M.M., and Walpole, J. Why the Grass May Not Be Greener on the Other Side: A Comparison of Locking and Transactional Memory. *Proceedings of the 4th Workshop on Programming Languages and Operating Systems (PLOS 2007)*, (October 2007)
- [13] Joglar, X., Planas, J. and Gil, M. OS Paradigms Adaption to Fit New Architectures. *Workshop on Interaction between Operating Systems and Computer Architecture* (June 2008)
- [14] Mars, J., Williams, D., Upton, D., Ghosh, S. and Hzelwood, K. A Reactive Unobtrusive Prefetcher for Multicore and Manycore Architectures. *In Proceedings of the Workshop on Software and Hardware Challenges of Manycore Platforms*, (June 2008), 41 – 51.
- [15] Zea, N., Sartori, J., and Kumar, R. Servo: A Programming Model for Many-core Computing. *In Workshop on Design, Architecture, and Simulation of Chip Multiprocessors(dasCMP)*, (December 2007)
- [16] Fredman, M., Wenisch, Th., Alamäki A., Falsafi B. and Moshovos, A. Temporal Instruction Fetch Streaming. *In International Symposium on Micro-architecture*, (November 2008)

- [17] Suleman, M.A., Mutlu, O., Qureshi, M.K., Patt, Y.N. Accelerating Critical Section Execution with Assymmetric Multi-Core Architectures. *In International Conference on Architectural Support for Programming Languages and Operating Systems*, (March 2009)

J-DSE: Joint Software and Hardware Design Space Exploration for Application Specific Processors

Marco Paolieri^{1,3}, Ivano Bonesana², Roberto Gioiosa¹, and Mateo Valero^{1,3}

¹ Barcelona Supercomputing Center,
Barcelona, Spain,
marco.paolieri@bsc.es

² SUPSI,

Lugano, Switzerland

³ DAC - Universitat Politècnica de Catalunya,
Barcelona, Spain

Abstract. Modern embedded systems' applications are characterized by a high demand of computing power but classical embedded systems' constraints (power, area, etc.) still hold. The solution to this new challenge comes from the use of Multi-Core in embedded systems. However, providing high performance with a limited power consumption is still an open problem especially considering the rapid increase of design complexity and manufacturing cost. The use of customizable cores (like FPGAs and ASIPs) that serve as accelerators is one of the most interesting solution for modern embedded systems because designers can benefit from the intrinsic capabilities of these accelerators: configurability and customizability.

In this paper we present a methodology that aims to reduce the effort of designing an application specific processor and improve the application's performance through a joint software-hardware design space exploration. Our methodology, named J-DSE, allows the designer to define different software optimizations as well as hardware configurations. A larger design space is explored through a genetic algorithm in order to find an optimal solution according to the user-defined cost function.

We applied the J-DSE methodology to the Imaging Pipeline application on the VEX simulation framework and obtained a performance improvement of 27% respect to a methodology that considers first the architecture and then the software implementation.

1 Introduction

The ever-increasing demand of computing resources of modern embedded systems' applications and the limited power budget available for these kind of systems have driven the market towards the use of Chip Multi-Core [1]. Several solutions have been already proposed to satisfy the application requirements in terms of performance, power consumption, chip area, etc., including the use of special-purpose, cost-effective cores (accelerators) [2]. A common example for the use of such MPSoC platform (e.g. the one shown in Figure 1) composed by CPUs, FPGAs and ASIPs is the software defined radio. These solutions perform critical operations in hardware guaranteeing high performance with limited power budget. However, obtaining high performance and maintain limited power consumption at the same time is still a challenging task, especially considering that the time-to-market for embedded applications has drastically reduced and the cost of the system design is constantly increasing. In order to guarantee re-usability and a short time-to-market, some of the most successful solutions consider the use of FPGAs or *Application Specific Instruction-set Processors* (ASIPs) [3] as hardware accelerators. In fact, both FPGAs and ASIPs provide some flexibility to the designers in terms of the functionalities implemented in hardware as shown in Figure 4.

In this paper, we focus on ASIP processors, more specifically on *Very Large Instruction Word* (VLIW) architectures, which represents one of the possible processor design style. An ASIP is a processor with the Instruction-set tailored to benefit a specific application providing a tradeoff between the flexibility of a general purpose CPU and the performance of an ASIC [3] (*Application-Specific Integrated Circuit*). The designer can *configure* (change architectural parameters like the number of execution units, memory ports, etc.) and *customize* (define special purpose instructions) a processor with the goal of finding the best solution in terms of a defined cost function. In VLIW processors the architecture is directly exposed to the compiler that targets the ILP, scheduling the execution of operations in parallel on the hardware.

Classical processor design methodologies only consider one degree of freedom when exploring the processor design space, namely the hardware architecture parameters. The best solution is evaluated according to a cost function defined by the designer. However, to better exploit the ILP of an application, software optimizations should also be taken into account, especially considering that the compiler needs the help of the programmer in order to achieve a high ILP [4]. Generally, software optimizations are carried out manually [4], thus, only a limited number of possibilities can be considered. Moreover, in this scenario the implementation of the application is fixed. We argue that, if the application is considered as part of the design process and different coding alternatives and optimizations are explored together with the architectural parameters, a better solution can be identified. This paper introduces our methodology and tools which work in

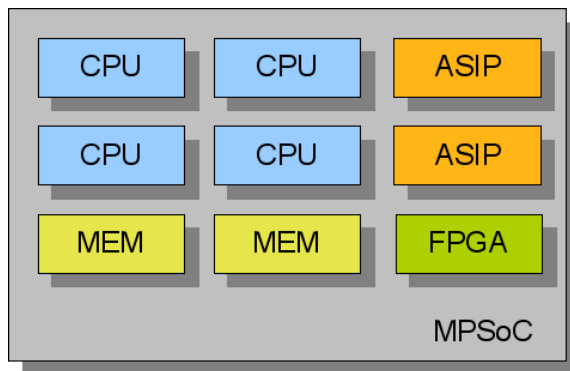


Fig. 1. An example of an MPSoC platform

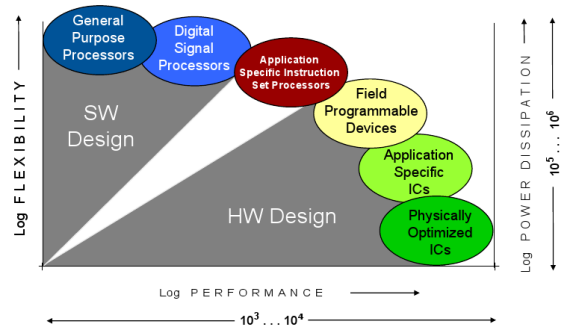


Fig. 2. ASIPs characteristics from[2]

a three dimensional space: the hardware, the application (code implementations, optimizations, customizations and compiler options) and the cost function. We call our methodology *J-DSE* for *Joint software and hardware Design Space Exploration*. Using our methodology we show, for the first time, how performance may be improved if alternative ways of coding, compiler options as well as architectural parameters, are considered simultaneously.

Our methodology consists of three steps:

1. Profiling the application in order to identify its computational intensive parts;
2. Defining the hardware (architecture parameters) and software (coding alternatives, code optimizations, custom instructions and the compiler options) design space.
3. Exploring the design space to identify the optimal solution according to the user defined cost function.

To provide evidence that our methodology works properly, we applied our approach to a typical image processing application: Imaging Pipeline. This benchmark allows us to compare our final optimal solution to the one found in [5]. We run our experiments on VEX [6] (*VLIW Example*), a compilation-simulation framework developed by HP Research Labs. It targets a wide class of VLIW processor architectures enabling compilation, simulation, analysis and evaluation of C programs for those architectures. VEX compiles a C source program generating the assembly instructions (.s file); then it builds the final simulator source code that is compiled using the host machine C compiler.

The rest of this paper is organized as follows: Section 2 describes our methodology; Section 3 provides information about related work; Section 4 shows how our methodology works for the chosen benchmark and provides experimental results; finally Section 5 addresses our conclusions and finalizes the paper.

2 Methodology

ASIPs are usually designed exploring a two dimensional space composed of the architectural parameters and the cost function defined by the user. We claim that that the *Design Space Exploration* (DSE) should also include a software dimension where different optimizations and possible implementations of the application are taken into account. Thus, our methodology performs a three-dimensional analysis.

Each software optimization can increase the performance of the application, although the combination of several optimizations simultaneously may reduce the overall performance. Even if some software optimizations (such as compiler options) can be explored automatically [4], in most of the cases those optimizations are tested manually and independently from the hardware design space exploration. Thus, the effect of different software optimizations on the possible architecture is not considered with classical methodologies. Moreover, different implementations of the same algorithm can have different performance when executed on different processors.

In our methodology the best solution is computed considering a bigger DSE. In fact software optimizations and architecture alternatives are considered when defining the design space. Figure 4 summarizes our methodology.

The side effect of enlarging the design space is that the number of points that need to be explored increases due to the addition of the new dimension in the space. Thus, more computational power or longer simulation time are required to perform an exhaustive exploration of the design space. To tackle this problem, we use a Genetic Algorithm for the exploration phase⁴.

Figure 3 shows a three-dimensional example space where our methodology works: optimizations are carried out on both hardware and software and the best solution, in terms of the user-defined cost function, is picked out. All the pareto-optimal point sets are identified and any of these optimal solutions can be selected according to design constraints: the dominated points are identified and discarded.

2.1 Profiling

Optimizing the application's source code so that it exposes a maximum level of ILP is crucial for our analysis, especially for VLIW processors, where the compiler is in charge of most of the optimizations. Luckily, in many cases a limited part of the application's source code accounts for the largest part of the execution time (for example 10% of the code takes 90% of the execution time). In order to identify these critical sections, we profile the application, both statically and dynamically. The former simply counts the instructions present in the assembly source code while the latter is able to count the number and the type of different operations

⁴ Other optimization algorithms can be considered

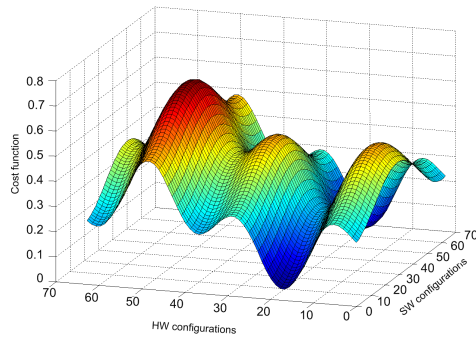


Fig. 3. The three dimensional analysis space

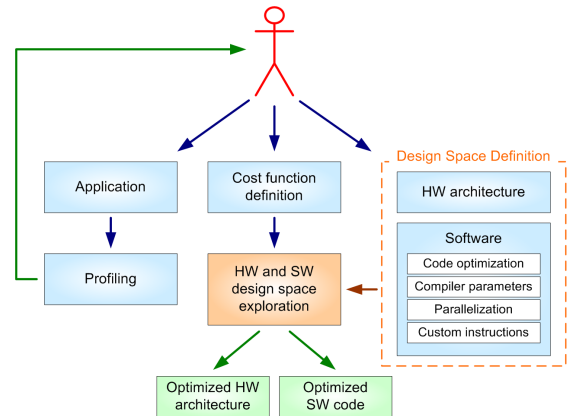


Fig. 4. Methodology flow

that are executed at run-time. Dynamic profiling at low level allows us to better define the architectural parameters we need to explore.

The VEX framework does not allow us to profile the application with a fine level of granularity (machine instructions level), hence, we developed an extension that serves this purpose. Considering that the assembly operations are implemented as C MACRO, we decided to design a Flex⁵ parser that takes as input the simulator source code, finds whenever an assembly instruction macro is called and increments a variable. When the simulation is finished, the final values of those variables are written to an output file. We decided not to build a parser with a fixed structure but to allow the users to define it as preferred. It is possible to change the assembly instructions that are profiled and easily group them according to specific criteria (e.g., group together all the ALU operations). This is exploited by means of an hash table listing the operations that have to be monitored. Each element of that table can be easily changed using some C MACROS. In case the cost function includes power consumption this feature can be very useful since information about the execution of different operations are required. We measured that the overhead of the dynamic profiling - whenever is enabled - corresponds to about 5% increase of the simulation time.

2.2 Software Optimizations

Profiling allows us to identify the bottlenecks of the application. With this information we are able to understand which architectural units are the most used and if and where the source code should eventually be optimized.

In the J-DSE methodology, software optimizations come in four forms:

1. *Coding Alternatives* A part of the algorithm can be coded in different ways, maintaining the correctness. Since VLIW processors rely on the compiler for the code optimizations, having an implementation of the application which enables the compiler to identify more ILP may result in a great performance improvement.
2. *Code Optimization* Regardless of how a specific part of the application has been coded, several code optimizations can be tested. For example, a loop can be unrolled by 2, 4, 8 times and, according to the

⁵ Flex is a fast lexical analyzer generator. It is a tool for generating programs that perform pattern-matching on text

number of functional units present in the processor, the performance of the application could improve or not. This is usually exploited by inserting `#pragmas`.

3. *Custom Instructions* The ISA of the processor can be extended with new, more performing instructions. The idea is that some very common operations can be performed directly by the hardware rather than with a sequence of assembly instructions (i.e., by software). Being able to identify and isolate those operations makes possible to greatly improve the performance of the application. Clearly, custom instructions do not come for free, the chip area would increase because of the need of new hardware resources.
4. *Compiler Optimizations* Different compiler optimizations can be tested for each of the implementation of the applications. As we said, the compiler plays a very important role with VLIW processors, and finding the correct set of compiler optimization parameters is crucial for the performance of the application.

Usually these software optimizations are introduced manually, although, in most of the cases, the application code is considered fixed and only a limited number of optimizations (such as compiler options or custom instructions) are carried on. The key point is that, even if independent software optimizations can improve the performance of the application, the usage of some of them simultaneously could as well decrease the application's performance [4]. As a result this activity is time consuming, hence, not all the possibilities can be explored manually.

Our tool, instead, allows the user to provide different software optimizations and automatically generates one implementation for each possible combination, defining the software design space. This part of the design space is merged with hardware design space coming out from the architectural configurations, i.e., each application implementation is coupled with each hardware configuration. In other words, the complete design space is explored in order to find the optimal solution.

We developed a preprocessing tool that allows the designer to insert at once all the different optimizations in the source code specifying the source code alternatives. The complete set is explored together with the architecture configurations. In this way it is not necessary to spend time manually modifying the source code after each test until an implementation that is considered good enough is found.

2.3 Design Space Exploration

Whenever a design space exploration (DSE) is performed the user must define the *cost function*, i.e., the function used to evaluate different possible solutions. Rather than explore exhaustively all the possible configurations, we can reduce the set of points in the space that the DSE algorithm must test with an optimization algorithm such as Genetic Algorithm [7], Simulated Annealing [8], etc.

We use a *Genetic Algorithm* (GA) to explore the design space. Genetic algorithms have been inspired by Charles Darwin's theory of evolution⁶. This algorithm [7] has been used to solve optimization problems in several application domains (like wire routing in electronic boards, etc.). We use GA to find the VEX software-hardware combination that achieves the best performance for our application.

Let us define a population as a set of individuals. Each individual is characterized by a set of parameters (*genes*) called *chromosome* and a value called *fitness*, which represents the value of the cost function. The goal of a GA is to maximize/minimize this fitness value⁷. The termination of the algorithm can be set by

⁶ This theory says that a population of living beings evolves itself to adapt to the environmental conditions. The evolution is based on the *natural selection* mechanism.

⁷ We can assume the fitness as the result of a cost function using as arguments the parameters contained in the chromosome.

the value of the fitness (e.g., invariance of the best individual, closeness to a target) or fixing the maximum number of generations produced. Once an initial population has been created and evaluated, it starts to evolve. The evolution is based on an operation called *reproduction* that can be divided into three steps: selection of the parents, combination of the chromosomes of the parents and mutation of some genes. The selection is a random based process that works on the values of the fitness of the solutions. In a maximization problem, the higher the value of the fitness, the higher the probability of a solution to be selected.

GAs are not optimal techniques. Since they are based on random choices, in certain cases they can fail or loop. These situations are avoided by a good choice of the parameters. Moreover, even though the solution found it is not guaranteed to be optimal, it is a good one.

3 Related Work

Several DSE methodologies for VLIW processors are available in the literature; some of them also take into account some software optimizations when exploring the design space. Niar et al. [4] describe a flexible framework for an adaptation of the micro-architecture and the compilation process. The use of a Compilation Option Space Exploration (COSE) makes their study similar to ours, though they only explore compiler options in their software/hardware DSE. This is a sub-case of our J-DSE methodology, because we also target other forms of software optimizations.

In [3], the authors describe an automated compilation flow that generates application-specific instructions. Their tool is orthogonal to ours and could be used on top of our profiling analysis to help the user to define the custom instructions.

In [9] a design space exploration for an Ogg/Vorbis decoder for VLIW architectures has been carried out: the methodology they use is an iterative procedure with subsequent refinements.

Ferrante et al. in [5] propose an example of VLIW architecture application-driven optimization using the VEX framework: they evaluate the optimal architecture according to the product area/performance cost function. In their work they do not use any algorithm to explore the design space but they perform exhaustive search on the defined space. They have a different approach respect to ours since they exploit software optimizations once they identified the optimal architecture. Our solution has the advantage that software optimizations can influence the resulting optimal architecture. In this paper, we use the same benchmark and cost function as in [5] and show how our solution, exploring simultaneously the software and hardware design space, finds a hardware architecture and a software implementation that achieve a performance improvement of 27%, in terms of cost function value, over the solution proposed by Ferrante et al. in [5].

Spacewalker [10] is a tool developed at HP Research Labs for automated design of a computer system for an embedded application, making a trade-off between cost and performance. It aims to create a set of good architectures for a given application. It does not carry out software optimizations and it is completely automatized.

Panis et al. in [11] presents a methodology for design space exploration of an embedded DSP core, adapting it to the application specific requirements. It can be used to understand the requirements of the application code on the processor architecture or fine-tune the chosen architecture.

In [12] the authors describe an approach for hardware/software design space exploration for reconfigurable processors. The authors extend the standard programming model to execute some critical parts of the application on an FPGA that has been added as a function unit of a RISC processor. The DSE is necessary to identify the most time-consuming portions of the code and execute them as single instruction on FPGA-based units.

EPIC-Explorer [13] is a framework to simulate a parameterized VLIW-based platform that will allow an embedded system designer to evaluate any instance of the platform in terms of performance, area and

power consumption. This paper does not present a methodology to perform design space exploration but a framework that provide the possibility to perform design space exploration on top of it. It could be possible, for instance, to apply our proposed methodology on such framework to optimize the VLIW-platform for a given application according to a defined cost-function.

Performance and flexibility are two key attributes of embedded systems design, the best performance is obtained from custom designed integrated circuits while the maximum flexibility by general purpose processors. A new design approach is presented in [14] where a partially re-configurable embedded processor rASIP is described.

4 Experimental Tests

This Section shows how the J-DSE methodology and tools can be used to design an application-specific VLIW processor.

In order to experiment our methodology, we used the Image Pipelining application as benchmark. This is the same benchmark used in [5], thus, we can compare the solution obtained by the use of our methodology to the one proposed by Ferrante et al.

The core of the benchmark is an algorithm that converts a *jpeg* image into a half-toned one by performing the following steps:

1. decompression of the jpeg image into RGB color space
2. resizing the RGB image
3. color space conversion from RGB to CMYK
4. dithering⁸ of the CMYK image to obtain the half-toned one⁹

The most computational intensive function of the benchmark is the one that combines together steps 3 and 4. For this reason we will concentrate on this part when doing the coding optimizations and defining the custom instructions. Clearly, compiler optimizations will affect the whole application.

4.1 Cost Function

During the DSE, the possible solutions present in the design space are evaluated with a user-defined *cost function*. In our experiments we used the same cost-function defined in [5]. Here it follows a brief description of that cost function:

- The Area-Performance product is taken into account:

$$C(A, T) = \frac{A}{A_{def}} \times \frac{T}{T_{def}} \quad (1)$$

where A is the chip-area of the architecture, and T is the amount of clock cycles necessary for the execution of the benchmark. A_{def} and T_{def} are values obtained on a *reference* architecture illustrated in Table 2.

- The number of clock cycles required for the execution is an output of the VEX simulator.

⁸ Dithering is a technique to create the illusion of color depth in images with a limited color palette. Colors not available are approximated by a diffusion of colored pixels from within the available palette.

⁹ The algorithm used for the dithering is the classic Floyd-Steinberg algorithm.

- The area is estimated through some approximations and expressed in *register bit equivalent area* (rbe).

$$A = A_{mem} + N_{cluster}(A_{reg} + N_{alu}A_{alu} + N_{mul}A_{mul}) \quad (2)$$

where N_{alu} is the number of ALUs, A_{alu} is the area of an ALU (1481 rbe), N_{mul} is the number of multipliers, A_{mul} is the area of a Multiplier (25600 rbe). The area occupied by the registers is obtained with the following formula:

$$A_{reg} = (N_{reg} + 3 \times P_{reg}) \times (32 + 3 \times P_{reg}) \quad (3)$$

where P_{reg} is the number of ports and N_{reg} is the number of registers in the register files. Finally, $N_{cluster}$ represents the number of clusters in the processor. The area on the chip occupied by the memory is computed as follows:

$$A_{mem} = 0.6 \left[8 + \frac{29 - \log_2 \frac{Size}{Ass}}{LineSize} \right] \times Size \quad (4)$$

$$\times [1 + 0.25(N_{memport} - 1)] \quad (5)$$

where $Size$ is the size of the cache in bytes, $LineSize$ is the size of the line (in bytes), Ass is the associativity and $N_{memport}$ is the number of memory ports. For a complete explanation of the formula please refer to [5].

4.2 Design Space

Table 2. Architectural parameter of the default machine

Table 1. Explored architectural configurations

Parameter	Range	Combinations
Number of Clusters	1,2,4	3
Issue Width (=n)	4,8,16,32	4
ALU per cluster	n	1
MemLoad(=p)	n/2,n	2
MemStore	p	1
MemPorts per cluster	p	1
Multiplier per cluster	2	1
DCache Size (Kbyte)	32,64,128	3
DCache Associativity	4,8	2
DCache Line Size (byte)	32,64	2
ICache Size (Kbyte)	32,64,128	3
ICache Associativity	1	1
ICache Line Size (byte)	64	1

Parameter	Value
Number of Clusters	1
Issue Width	4
Data Cache Ports	1
Register Files	2
General Purpose Registers (32 bit)	64
Branch Register (1 bit)	8
Integer ALU	4
Integer Multiplier	2
DCache Size	32 Kbyte
DCache Associativity	4
DCache Line Size	32 byte
DCache Miss Penalty	36 cycles
ICache Size	32 Kbytes
ICache Associativity	1
ICache Line Size	64 byte
ICache Miss Penalty	45 cycles
T_{ref}	268.662 msec
A_{ref}	81993

Hardware Design Space The VEX framework allows us to customize an architecture using a set of parameters. Among all the possible parameters, we chose those shown in Table 1 for our design space. We based the definition of the parameters for the architecture configurations on the results of dynamic profiling

(shown in Figure 5). The dynamic profiling was carried out using the extension we developed on the VEX simulation framework. It is possible to notice how the most executed operations of the algorithm are ALU and memory operations. These results are close to the ones obtained with the static profiling[5] but since they consider the dynamic flow of the instructions they are much more meaningful.

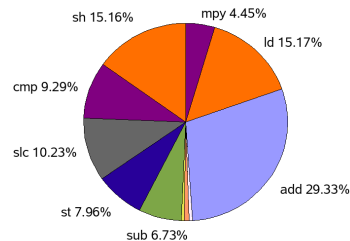


Fig. 5. Dynamic Profiling of the Image Pipelining Benchmark

We tested an increasing number of ALUs from 4 to 32 (for each cluster). A small amount of multiplications was reported by the profiler, so we left the number of multiplier units fixed to the minimum allowed by VEX.

As shown by the results, memory accesses are critical, hence, we had to explore several memory port configurations for loading and storing data. We also modified the total cache size, the associativity and the line size.

Since the control flow of the algorithm is very regular, branches and control flow operations are not critical for such application.

Software Design Space The other dimension explored during the design space exploration is the software. As already described in [4] even if independent software optimizations can improve the performance of the application, the usage of some of them simultaneously could as well decrease the application's performance. As already described we created a tool that pre-process the source code and generates a new software implementation (according to the parameters that are being passed). This tool is executed during the design space exploration to couple a given architecture with a given software implementation.

Let us provide a simple example of how to use the tool: if the programmer identify a critical part of the source code that can be optimized in different ways, or simply he wants to analyze how different coding alternative reflect on the final performance he can code as follows:

```

###START_BLOCK 1
***START_ALTERNATIVE 0
// C code of alternative 0
***END_ALTERNATIVE 0

***START_ALTERNATIVE 1
// C code of alternative 1
***END_ALTERNATIVE 1

***START_ALTERNATIVE 2

```

Table 3. Software Design Space

Parameter	Value
Loop Unrolling	4,8,16
Custom Instructions	0,1,2

```
// C code of alternative 2
***END_ALTERNATIVE 2
***END_BLOCK 1
```

The user defines a block of code that can be coded in different ways using the tags `***START_BLOCK` and `***END_BLOCK`. Inside this block the user can specify a variable number of coding alternatives by defining several `***START_ALTERNATIVE` and `***END_ALTERNATIVE` tags. By running our tool just one of this alternatives is selected every time and tested. Of course any .C or .h file can contain any number of blocks.

For the Image Pipelining benchmark we identified a lot of loops and repeating instruction-patterns so we explored the possibility of unrolling loops by different amounts: by 4,8,16 and we tested implementations with or without two different type of custom instructions.

The custom instructions we identified are the followings:

```
dx_c = _DITH_ALL(ex_c,eym_c,ey_c,eyp_c);
```

that performs in hardware the C instruction

```
(ex_c * 7 + eym_c + ey_c * 5
+ eyp_c * 3) / 16;
```

We also tried to split this C code into two custom instructions, such that

```
dx_c = (_DITH_75(ex_c,ey_c)
+ _DITH_31(eyp_c,eym_c)) / 16;
```

In order to compare the results with the ones described in [5] the additional area required in hardware to executed the custom instructions is not taken into account in the cost function. This cost should be considered in a more careful analysis. The space explored on the software domain is summarized in Table 3

4.3 Design Space Exploration

After the definition of the *joint* design space - that contains 7776 different configurations - we performed both an exploration based on the GA and an exhaustive search (in order to validate our results).

The GA required less than 2000 simulations (reducing the simulation time by more than two thirds) and found a configuration with a cost-function value of 0.3577 while the best configuration computed with the exhaustive search has a value of 0.355. This proves how the GA represents a very good choice and it differs from the best solution only by 0.665%.

Table4 shows the value of the optimal configuration, while Figure 6 shows the 3D space that has been explored by our methodology in order to define the best configuration for the Imaging Pipeline benchmark. The optimal configuration results to perform 65% better than the reference one (in terms of cost function). Compared to the optimal configuration found in [5] by Ferrante et al. our solutions results to be 27% better¹⁰ This clearly shows how the software and hardware solutions should be coupled together during the design space exploration in order to achieve better performance.

¹⁰ We considered the same parameters, obtaining a cost-function value of 0.48, while our best configuration is 0.35. Please remind that according to the definition of the cost function a smaller value represents a better solution.

Table 4. Optimal configuration discovered by J-DSE methodology

Parameter	Range
Number of Clusters	1
Issue Width	4
ALU per cluster	4
MemLoad	2
MemStore	2
MemPorts per cluster	2
Multiplier per cluster	2
DCache Size (Kbyte)	32
DCache Associativity	4
DCache Line Size (byte)	32
ICache Size (Kbyte)	128
ICache Associativity	1
ICache Line Size (byte)	64
Loop Unrolling	4
Custom Instructions	1

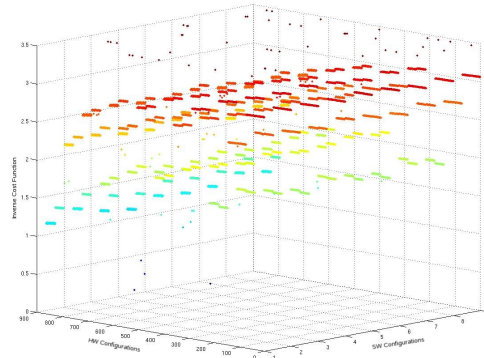


Fig. 6. The three dimensional analyzed for Imaging Pipelining benchmark

5 Conclusion

The common trend of embedded system designers is to move towards CMPs equipped with special cores that accelerate critical functions and satisfy the application requirements in terms of performance, power consumption, chip area, etc. With this solution, the complexity of the software increases, hence, most of current embedded solutions use customizable accelerators like VLIW and ASIP. Even if those accelerators are usually configured and customized for specific functionalities, their flexibility reduce the time-to-market and the design cost.

Common design methodologies try to identify the best architecture based on a design space exploration (i.e., carried out changing architectural parameters) and, after some preliminary tests, minor software optimizations are performed. In this paper we present J-DSE: a methodology that considers simultaneously the software and hardware design space. In this way different software implementations are directly coupled together with different hardware configurations. In terms of cost function our results show an improvement of 27% respect to a classic methodology and almost 65% improvement respect to the reference configuration.

Future works could involve testing this methodology on different benchmarks defining a bigger software design space. We also plan to apply such methodology to exploit the function-level parallelism, allowing the possibility to select functions that could be executed in parallel, which perfectly fits modern architecture (e.g., IBM Cell [15]).

References

1. A. Ltd., "Arm11 mpcore processor technical reference manual," 2005.
2. R. Leupers, "From ASIP to MPSoC architectures and design tools for communication and multimedia systems,"
3. J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *Proc. on the International Symposium on FPGAs*, Feb. 2004.
4. S. Niar, N. Inglart, M. Chaker, S. Hanafi, and N. Benameur, "Facse: a framework for architecture and compilation space exploration," in *Proc. on DTIS*, Sept. 2007.
5. A. Ferrante, G. Piscopo, and S. Scaldaferri, "Application-driven optimization of VLIW architectures: a Hardware-Software approach," in *Proc. on RTAS*, March 2005.
6. P. Faraboschi, *The VEX System*. HP Research Labs, 2004.
7. J. H. Holland, *Adaptation in natural and artificial system*. The University of Michigan Press, 1975.
8. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, Number 4598, 13 May 1983, vol. 220, 4598, pp. 671–680, 1983.
9. F. Ferrari and E. Amador, "Design exploration for an Ogg/Vorbis decoder for VLIW architectures," in *Proc. on WASP*, Oct. 2007.
10. G. Snider, *Spacewalker: Automated Design Space Exploration for Embedded Computer Systems*. HP Research Labs, 2001.
11. C. Panis, U. Hirschrott, G. Laure, W. Lazian, and J. Nurmi, "DSPxPplore - design space exploration methodology for an embedded DSP core," in *Proc. on SAC*, Mar. 2004.
12. A. L. Rosa, L. Lavagno, and C. Passerone, "Hardware/software design space exploration for a reconfigurable processor," in *Proc. on DATE*, Mar. 2003.
13. G. Ascia, V. Catania, M. Palesi, and D. Patti, "EPIC-Explorer: a parameterized VLIW-based platform framework for design space exploration," in *Workshop on ESTIMedia*, Oct. 2003.
14. A. Chattopadhyay, W. Ahmed, K. Karuri, D. Kammler, R. Leupers, G. Ascheid, and H. Meyr, "Design space exploration of partially re-configurable embedded processors," in *Proc. on DATE*, Mar. 2007.
15. M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic processing in Cell's multicore architecture," *IEEE Micro*, Mar. 2006. Available from http://www.research.ibm.com/people/m/mikeg/papers/2006_ieeemicro.pdf.

Building a Java MapReduce Framework for Multi-core Architectures

George Kooor, Jeremy Singer and Mikel Luján
Advanced Processor Technologies Group
The University of Manchester, UK

Abstract. MapReduce is a programming pattern that has been proved to be a simple abstraction on top of which can be built an efficient platform for large-scale data processing in distributed environments, such as Google or Hadoop. With this pattern, application logic is expressed using sequential *map* and *reduce* functions. Thus, a runtime system can exploit the lack of side effects (*pure functions*) in these functions to execute concurrently. The runtime framework also takes care of the low-level parallelisation and scheduling details. The success of the MapReduce pattern has led to several implementations for various scenarios. This paper introduces MR-J, a MapReduce Java framework for multi-core architectures, and reports the scalability results from the first experiments.

Keywords: MapReduce, parallel software framework.

1 Introduction

The MapReduce programming pattern was, by no means, invented by Google. Its roots can be traced back to functional programming [1]. Nonetheless, it has attracted a fair amount of attention from industry, academics and open-source projects (Hadoop [3]), since Google made public [2] that in their experience this pattern was easy to use and provided a highly effective means of attaining massive parallelism in large data-centers. The pattern is not a silver bullet that can be applied to any general-purpose application (some consider it a step backwards [8]), but it covers an important part of the application spectrum. Our objective is to investigate the MapReduce pattern in the context of multi-core architectures and not within data-centers as commonly used by Amazon, Facebook, Google and Yahoo, to name a few [5].

We have selected Java as the programming language because the main open-source implementation of MapReduce, *i.e.* Hadoop, is also developed in this language. Thus, our efforts and findings can contribute directly to the evolution of Hadoop. These efforts have produced MR-J, a MapReduce Java framework for multi-core architectures, and this paper reports the results from the first experiments.

The Phoenix framework [6, 7] is the only other MapReduce framework that targets multi-core architectures. Since Phoenix is implemented in C, the implementation can take advantage of pointers (*e.g.*, to avoid copying data) in ways that are not feasible within Hadoop or MR-J. Thus, the contribution of this paper is to report for the first

time the scalability of 4 benchmarks implemented with MapReduce in Java on a small multi-core architecture.

The paper is organised as follows. Section 2 presents the background on MapReduce and a brief description of its implementation on clusters or data-centers. Section 3 takes a closer look at a MapReduce implementation focused on a single chip; specially Phoenix. Section 4 describes the implementation of MR-J based around a divide-and-conquer approach. Section 5 shows the results from our first experiments on a small multi-core (Intel core i7) system using 4 different benchmarks. Finally Section 6 summarises MR-J and its first performance evaluation.

2 Background on MapReduce

The primary advantage of using the MapReduce pattern is that the application logic is expressed using sequential *map* and *reduce* functions. Grounded on functional programming, these functions must not have side effects; *i.e.* must be *pure functions*. Thus, a runtime system can exploit this property and execute the functions concurrently. MapReduce facilitates the automatic parallelisation of applications through the guarantees provided by the functional programming construct.

The `map` function takes `<key, value>` pair as input and emits an intermediate `<key, value>` pair as output. The inputs to the `map` and `reduce` functions are processed independently without any dependency on other elements of data, thereby avoiding the need for synchronisation and contention for shared data. In general, the majority of the frameworks implementing the MapReduce pattern has at least `map`, `merge` and `reduce` phases. In the `map` phase, the `map` function is executed in parallel by different worker threads as `map` subtasks. The output from each `map` subtask is written to a local data structure; such as arrays, lists, or queues. The execution of the `map` phase is followed by the `merge` phase. In this phase the runtime system combines the intermediate `<key, value>` pair output from each `map` subtask so that values from the same keys are grouped together to form a unique `<key, value>` pair. The framework partitions the unique `<key, value>` pairs among the `reduce` task workers. As with the `map` task, the `reduce` tasks are executed in parallel without any dependencies on other elements. The `reduce` task usually performs some kind of reduction operation such as summation, sorting and merging operations. The output from each `reduce` task worker is written to either a distributed file system in the case of cluster-based implementations, or it is written to a local data structure, which is then merged to produce a single output, in the case of multi-core architectures.

Cluster-based implementations of MapReduce (such as Google's and Hadoop) normally target large data-centers using commodity systems interconnected using high-speed Ethernet networks. These implementations rely on specialised distributed file systems such as Google's GFS [GFS] and Hadoop's HDFS to manage data across the distributed network. The implemented runtime system spawns worker threads on each node in the cluster. Each of these worker threads is assigned either a `map` or a `reduce` task. Only one of the worker threads is elected to be the master. Each job consists of a set of `map` and `reduce` tasks along with the input data. The runtime system automatically partitions the input data based on a *splitting* function into

smaller partitions or *chunks*. The selection of the chunk size is based on the block size of the distributed file system used by the framework. In the case of Google a default chunk size of 16MB to 64MB is used, whereas in the case of Hadoop a default chunk size of 64MB is used.

The master worker assigns these partitioned data to map task workers, which are distributed among the cluster nodes. Each map task worker processes the input data in parallel without any dependency. The output from the map task worker is stored locally on the node on which the task is executing. Once all the map tasks are completed, the runtime system automatically sorts the output from each map task worker so that the values from the same intermediate keys are grouped together before it is assigned to the reduce task worker. The reduce tasks are assigned to the reduce task workers by partitioning the intermediate sorted keys using the partitioning function. The default partitioning function used in both Google's model and Hadoop is based on key hashing: $\text{hash}(\text{key}) \bmod R$, where R is the number of reduce task workers.

In a distributed environment the master is responsible for the following tasks. It maintains the status information and identity for map and reduce tasks. It is responsible for transferring the location of the file from the map task worker to the reduce task worker. It delegates map or reduce tasks to each worker, maintains locality by assigning tasks locally to the workers and manages the termination of each worker. An essential feature for any cluster-based implementation is its ability to detect failures and slow tasks during execution. This is important because machine failures can be frequent due to the large cluster size. Both Google's framework and Hadoop implement effective fault-tolerance mechanisms that can detect slow nodes and node failures.

3 MapReduce on multi-cores

Implementations of MapReduce that are not targetting clusters have started appearing since 2007. For example, He *et al.* [9] and Kruijf *et al.* [10] have developed implementations for GPGPUs and Cell processors, respectively. The Phoenix project [6, 7] is the only previous implementation that focuses on shared memory multi-core architectures. Accordingly we discuss this implementation at greater length. The underlying principle of Phoenix is based on Google's MapReduce framework; hence they share several features. Phoenix is implemented in C and P-threads. The major difference when comparing Phoenix with Google's framework is that it uses threads to do the processing instead of worker nodes and relies on inter thread communication instead of remote procedure calls [1]. The fundamental structure of the Phoenix API resembles Google's MapReduce API, as both maintain a narrow and simplified interface. The Phoenix runtime system creates and manages threads across multiple cores, dynamically schedules map and reduce tasks to the worker threads, handles communication among worker threads and maintains state for each worker thread.

Figure 1 summarises the execution workflow within Phoenix. The user program initialises the scheduler by invoking an initialisation function. The scheduler spawns several worker threads according to the number of cores supported by the multi-core

chip. Each worker thread is assigned map or reduce tasks by the scheduler. The scheduler partitions the input data using the splitter function. The default splitter function uses the cache sizes of the chip to determine the chunk size. Partitioned data is forwarded to the map task workers assigned dynamically by the scheduler. Once the map task worker completes processing the input data, it stores the output in a buffer. It also merges the values from the same key to form a unique key for each set of values. Reduce tasks start only when the entire map task completes. Reduce tasks start only when the entire map task completes.

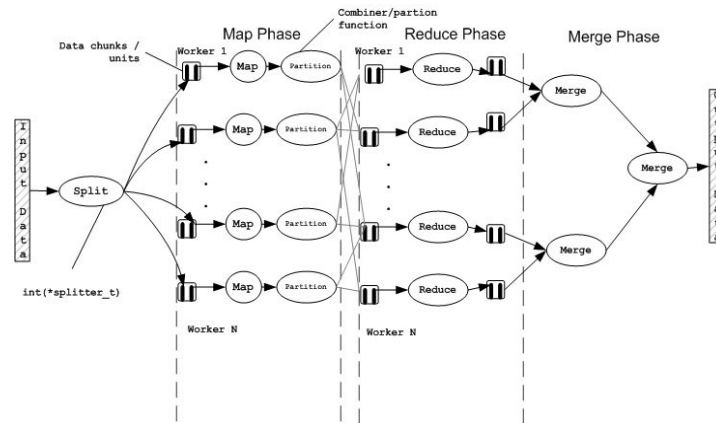


Figure 1 Phoenix execution overview (image adapted from [6]).

In the reduce phase the reduce tasks are assigned dynamically by the scheduler based on the partitioning function. The reduce task worker computes the data from the intermediate map queue. The output from the reduce task worker is stored in the local reduce queue to minimise the contention.

The merge phase is considered optional. In this phase the output from the reduce queue is merged together to produce a single output. In Google's implementation, this phase is achieved by making a recursive call to MapReduce. Ranger *et al.* [6] have reported that a merge phase is not required for most of their benchmarks and the overhead associated with it is comparatively lower than making recursive calls to MapReduce.

The default splitting function to partition the input data uses the L1 data cache size to determine a good chunk size of the partition. Phoenix also provides an option for developers to specify a custom partitioning function. This custom function can take advantage of application knowledge and avoid problems with load imbalance; keys can be associated with different values, the computation required by various keys may differ resulting in load imbalance.

Phoenix detects faults through a timeout mechanism. It uses execution time of a similar task as a measure to determine the maximum duration required to execute a task. Limitations of the fault-tolerance mechanism implemented in Phoenix system are identified as follows; lack of support to determine if a task is completed successfully by a worker thread and single point of failure for the scheduler.

To sum up, Phoenix is a mature and sophisticated software that is in its second public release. The implementation of MR-J is similar in many aspects, but we make no attempt, for the time being, to provide fault-tolerance mechanisms. We are focusing on evaluating whether a Java implementation of MapReduce can scale. In other words, we want to understand whether not having the same low level of control as in C will affect scalability.

4 Overview of MR-J

The design of MR-J shares many features with Hadoop at the application interface level (and also, *e.g.*, for job submission, setting the configuration parameters, and initialising the framework) as both frameworks are implemented in Java. In contrast to Hadoop, the current implementation of MR-J is designed specifically for multi-core architectures; as a result, the execution flow is closer to Phoenix.

Comparing the application interface with Phoenix, the WordCount application (see description in Table 1) requires the programmer to provide implementation for four user-defined functions (`map`, `reduce`, `splitter`, and `keycmp`). On the other hand, MR-J requires only the first two functions.

The distinguishing feature of MR-J is that it exploits a recursive divide-and-conquer approach. The implementation takes advantage of this by relying on work stealing and the Java fork-join framework (part of pre-release version of `java.util.concurrent` package for JDK1.7).

The sequence diagrams illustrated in Figure 2 and 3 provide an execution overview of the map and reduce phases. UML2 notations for parallel combined fragments (highlighted box in the diagrams) are used to indicate fork/join parallel executions. Partitioning of the input data creates subtasks with equal size of partitioned data units. Worker threads in the fork-join pool execute the generated subtasks. The number of worker threads initialised in the pool is a parameter of the framework and, normally, corresponds to the number of cores available. Thus, MR-J generates tasks dynamically and has a flexible mechanism to control the granularity of subtasks.

In order to improve performance for recursive calls to the `map` and `reduce` functions, a job-chaining functionality, similar to Hadoop's, has been implemented in MR-J. Job chaining enables multiple calls to map and reduce phases during a single MapReduce execution. This feature is required, for example, to implement the Kmeans benchmark and is not part of Phoenix where the runtime iterates through the map and reduce phases to compute the final cluster for a given set of coordinates. The benefit of using the job-chaining feature is that all the worker threads and data structures created during the first map and reduce phases are reused.

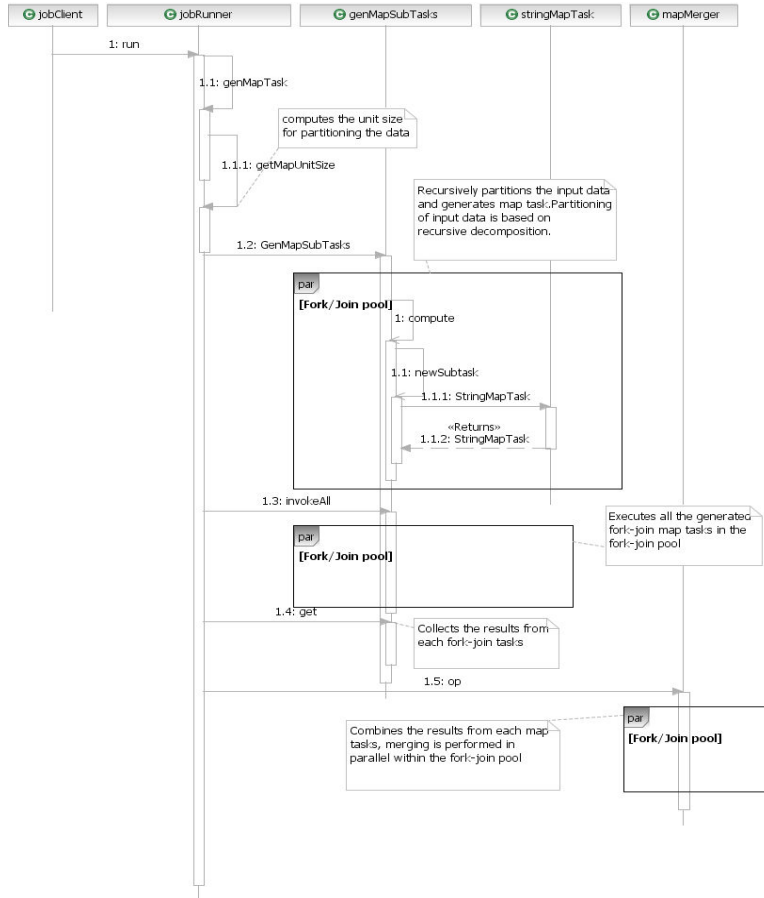


Figure 2 Sequence diagram for the map phase in MR-J.

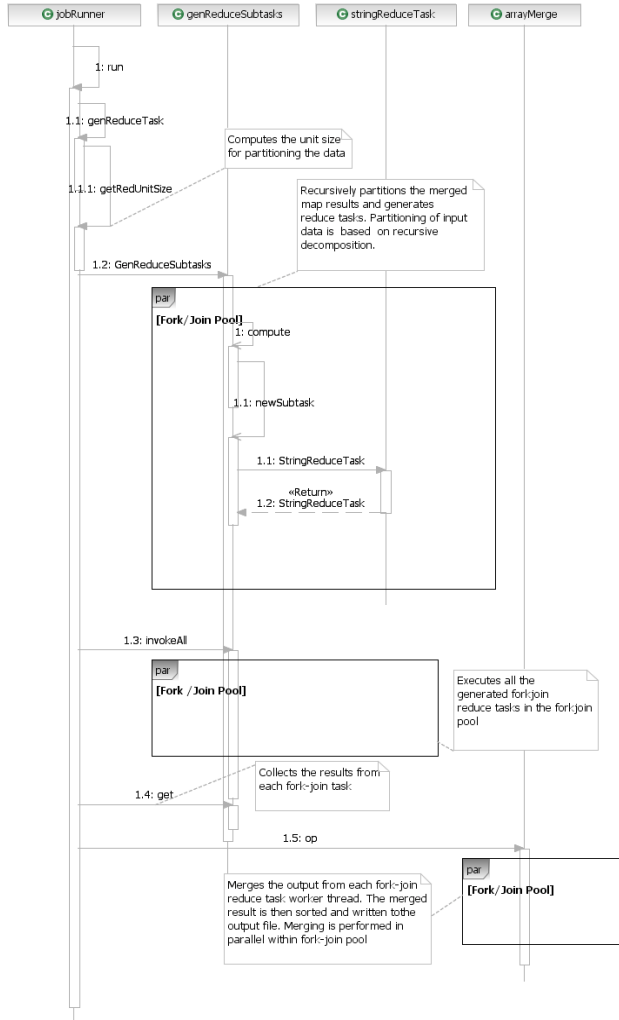


Figure 3 Sequence diagram for the reduce phase in MR-J.

5 Evaluation

The experiments consider the scalability of the MR-J framework and also examine the execution time breakdown of the different phases of execution of MapReduce.

5.1 Experimental Setup

Experiments are performed on a small multi-core system with one Intel Core i7 processor (*i.e.* four cores, 2 hyper-threads per core) running at 2.6GHz with 6GB of memory. The machine has an OpenSuse Linux 11.1 installation with Sun JVM version 1.6 (build 14.0-b08) using a fixed size heap of 4GB. Each experiment is executed five times and the average time is used in the results. We use the nanosecond resolution timer and the speedup is calculated according to T_1/T_p , where T_1 refers to the execution time using MR-J on a single thread and T_p refers to the parallel execution time on p threads. For all the experiments MR-J is configured so that it generates the same number of map tasks as the number of threads in the pool. The number of reduce tasks is adjusted dynamically based on the estimates of the reduce task cost and the number is always equal or less than the number of threads in the pool.

Table 2 presents a summary of the benchmarks we have implemented and the datasets used in the experiments. These benchmarks have been used for the evaluation of Phoenix as well as other MapReduce frameworks. It follows a brief description of how each benchmark is implemented within the MapReduce framework:

WordCount: Counts the number of times each word is repeated in the input document (text file). Each map task operates on different chunks of input data (text file). It reads each line from the input chunk and emits an intermediate `<key, value>` pair, where key is a word in the line and value is a number assigned to each word. Implementation of map function uses `StringTokenizer` to extract the word from each line based on whitespaces (*i.e.*, default delimiter). Reduce task sums the values associated with the word.

Grep: Extracts a given regular expression pattern from the input file and counts the occurrence of the pattern in each line. Grep is implemented in a similar way to WordCount with some minor changes to the implementation of the map function. The map function in Grep makes use of `java.util.regex.Matcher` to extract `<key, value>` pairs, where key represents a matching pattern and value is a number associated with it. The reduce task sums the values for each matching pattern together. As with the WordCount application, in Grep a map function is called for each line in the input document and reduce function is called for each unique key produced by the map tasks.

Kmeans: Groups a set of coordinates to the nearest clusters (K). The algorithm mainly consists of the following tasks: to determine the centre for each cluster, to calculate the distance from each coordinate to the clusters, and to assign the coordinates to the nearest cluster. Since the computation performed in Kmeans is iterative, the implementation uses job chaining, a feature discussed in Section 4. The coordinates are partitioned dynamically among the map tasks. Each map task computes the distance from each set of allocated coordinates to the clusters and identifies the nearest cluster for a given coordinate. The `<key, value>` pair output from the map tasks consists of a cluster index as key and an index of the coordinate as value. Reduce tasks are executed for each cluster updating the cluster and recalculating the centroid for each cluster. This process continues until all the coordinates converge to the nearest cluster.

Matrix-Multiply: For given matrices A, B, and C, the matrix B is partitioned column wise (along the second dimension) among the map tasks. Each map task computes the result of the corresponding column in C, using a block partition of matrix B and reference to all elements in matrix A. The reduce phase is not required.

Each of the benchmarks provides an optional argument to write the results to an output file. The output results from the benchmarks are verified by comparing it with the sequential version.

Name	Description	Data-Set		
		Small	Medium	Large
WordCount	Counts the frequency of words in a file.	10MB	50MB	100MB
Grep	Extracts a given regex pattern from the input file.	10MB	50MB	100MB
Kmeans	Clustering algorithm for 3D data points	10K Points	50K Points	100K Points
Matrix-Multiply	Performs integer matrix multiplication.	512x512	1024x1024	4096x4096

Table 1 Summary of the benchmarks used in the experiments.

5.2 Scalability

Figures 4, 5 and 6 show the speedup graphs for the three different datasets. Immediately we can observe that the benchmarks can be separated into two groups. The speedup curve for Grep and WordCount presents good scalability reaching their peak values with 6 threads. For the small dataset, the speedups are above 5.5, while for the other two datasets they are above 10 and 12. With more than 6 threads, the interference between the JVM threads, OS threads and the application itself prevent higher improvements. The hyper-threading mechanism shows improvements when executing 4 to 6 threads.

The speedup curves for Kmeans and Matrix-Multiply are more modest reaching only values between 2 and 4. For all the datasets, these two curves are flatter, specially with more than 6 threads.

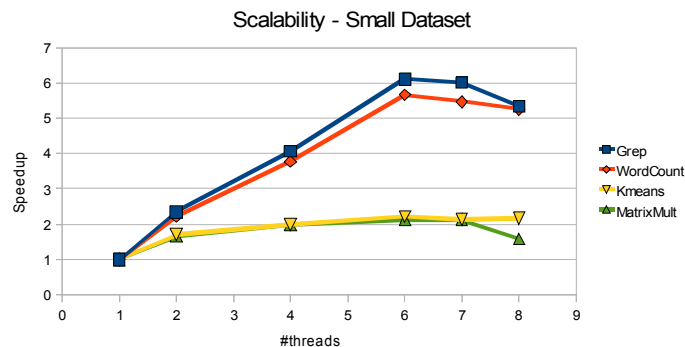


Figure 4 Scalability results with the small datasets.

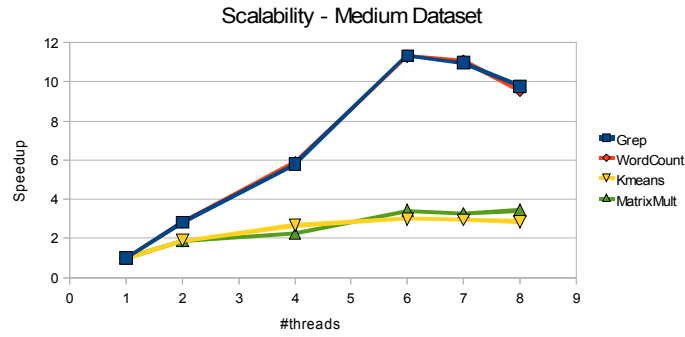


Figure 5 Scalability results with the medium datasets.

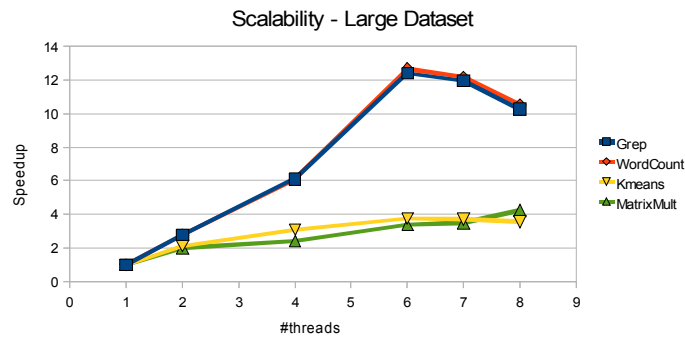


Figure 6 Scalability results with the large datasets.

5.3 Execution Time Breakdown

Figure 7 provides the execution time breakdown for the four benchmarks, but only for the large datasets. Note that the Matrix-Multiply benchmark only executes the map phase. The vertical axis (Y-axis) represents the normalised execution time relative to the map task execution time using only one thread. The horizontal axis denotes the number of threads; each benchmark application is grouped. The times for the map and reduce phases include the time required to generate and execute the Map and Reduce tasks. The time is clearly dominated by the merge phase. Kmeans is the only benchmark that exhibits noticeable reduce and merge phases.

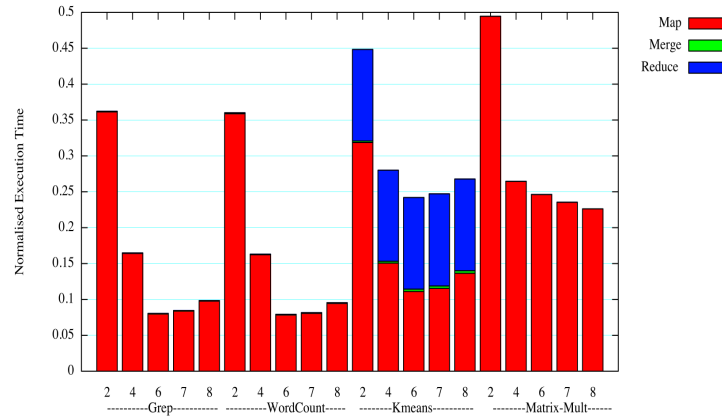


Figure 7 Execution time breakdowns for the large datasets.

6. Summary

MapReduce is a programming pattern that has been proved to be a simple and efficient platform for large-scale data processing in distributed environments, such as Google or Hadoop. Since 2007, implementations of MapReduce that are not targeting clusters or data-centers have started to appear. For example, He *et al.* [9] and Kruijff *et al.* [10] have developed implementations for GPGPUs and Cell processors, respectively. The Phoenix project [6, 7] is the only previous implementation that focuses on shared memory multi-core architectures.

MR-J is the only Java MapReduce framework that targets these multi-core architectures and shares similarities with Phoenix and Hadoop. However, a distinguishing feature is that MR-J is designed around a divide-and-conquer approach. The contribution of this paper is to report its first scalability analysis on an Intel core i7 system. We have shown that for two of the benchmarks MR-J is able to reach above 5 times speedup (maximum more than 12 times) over the T_1 using up to 8 threads. For the remaining two benchmarks scalability is more modest reaching speedup values between 2 and 4. Looking at the execution time breakdown, the most demanding phase is map, which dominates on all the benchmarks.

As future work, we want to investigate the impact of task granularity and the work-stealing behaviour on larger multi-core architectures. We also plan on increasing the number of benchmarks ported to MR-J.

Acknowledgments. Dr. Mikel Luján is supported by a Royal Society University Research Fellowship.

References

1. J. Dean, and S. Ghemawat, "MapReduce: simplified data processing on large clusters" *Communications of the ACM*, vol. 51, no. 1, pp. 107-113, 2008.
2. J. Dean, and S. Ghemawat, "MapReduce: simplified data processing on large clusters" In *Proceedings of the 6th Symposium on Operating Systems Design & Implementation*, pp. 137-150, 2004.
3. "Hadoop: Open source implementation of MapReduce", <http://lucene.apache.org/hadoop/>
4. T. White, "Hadoop: The Definitive Guide", O'Reilly, 2009.
5. Organizations using Hadoop. <http://wiki.apache.org/hadoop/PoweredBy>
6. C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems" In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 13-24, 2007,.
7. R.M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix Rebirth: Scalable MapReduce on a NUMA System" In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, pp. 198-207, 2009.
8. D.J. DeWitt, and M. Stonebraker, "MapReduce: A major step backwards" *The Database Column*, 2008.
9. B. He, W. Fang, Q. Luo, N.K. Govindaraju, and T. Wang, "Mars: a MapReduce framework on graphics processors" In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 260-269, 2008.
10. M. de Kruijf and K. Sankaralingam, "MapReduce for the Cell B.E. Architecture" *IBM Journal of Research and Development*, 53(5), 2009.