Running Legacy Applications on Multicore Machines

Ivo Anjo
INESC-ID/IST
ivo.anjo@ist.utl.pt

João Cachopo
INESC-ID/IST
joao.cachopo@inesc-id.pt

Jan 2009
Abstract

Multicore processors, capable of running multiple hardware threads concurrently, are seen by many as the future, and are becoming common on servers, desktops, laptops, and even smaller systems.

Unfortunately, most of the time these new machines are underutilized, as most current software is not written to take advantage of multiple processors. Also, with these new machines, more cores do not translate to more sequential performance, and legacy applications will not speed up by moving to a multicore.

This work explores speculative parallelization of these legacy applications, supported by a software transactional memory, with the objective of speeding up execution of sequential applications on parallel machines.

**Keywords:** Speculative Execution, Transactions, Transactional Memory, Legacy Applications, Multicore Architectures.
Running Legacy Applications on Multicore Machines *

Ivo Anjo and João Cachopo

ESW
INESC-ID Lisboa/Instituto Superior Técnico/Universidade Técnica de Lisboa
Rua Alves Redol 9, 1000-029 Lisboa, Portugal

Abstract. Multicore processors, capable of running multiple hardware threads concurrently, are seen by many as the future, and are becoming common on servers, desktops, laptops, and even smaller systems.
Unfortunately, most of the time these new machines are underutilized, as most current software is not written to take advantage of multiple processors. Also, with these new machines, more cores do not translate to more sequential performance, and legacy applications will not speed up by moving to a multicore.
This work explores speculative parallelization of these legacy applications, supported by a software transactional memory, with the objective of speeding up execution of sequential applications on parallel machines.

Key words: Speculative Execution, Transactions, Transactional Memory, Legacy Applications, Multicore Architectures

1 Introduction

The transition to multicore architectures is ongoing. Chip designers are no longer racing to design the fastest uniprocessor, instead turning to parallel architectures, with newer designs incorporating multiple cores, and capable of running many threads simultaneously.

The full power of these multicore chips is unlocked only when all cores are busy executing code. Yet, most desktop applications fail to take advantage of these processors, having little, if any, parallelism.

The problem for legacy applications is that for years new, faster, uniprocessors brought "free" speedups for all applications: just upgrade to the latest processor and your application goes faster, with no other modifications needed; this does not apply with multicore processors.

Concurrent programming is, unfortunately, harder than sequential programming. Moreover, even if newly developed applications are written with multicore architectures in mind, most of the already developed code is still sequential and it is not feasible to rewrite it within a reasonable time frame.

* This work is being performed in the scope of the Pastramy project (PTDC/EIA/72405/2006).
So, the problem is that we cannot expect to get better performance on most legacy applications just by upgrading to a multicore processor, and rewriting or adapting legacy applications poses many problems.

This work explores speculative parallelization of these legacy applications as a way of continuing to extract more performance for these applications on multicore architectures.

The remainder of this work is organized as follows. Section 2 gives an overview of the different approaches to legacy application parallelization, and introduces the goals for this work. Section 3 presents important research related to this work. Section 4 shows the architecture of the proposed solution. Section 5 explains how the work shall be evaluated, and finally Section 6 summarizes the current findings and proposed work.

2 Work Objectives

An application may be parallelized either by replacing used libraries by parallel implementations of those libraries, or by parallelizing the application itself.

Modern applications depend heavily on multiple shared libraries and frameworks, the application itself almost just providing parameters and glue to stick these components together. This arrangement makes common shared libraries a promising target for parallel implementation, as by parallelizing these libraries we get speedups that reflect on all the applications that use them.

This approach, although valid, has the downside of having to parallelize each library being used, and that only the applications using it benefit from the speedup; it does not solve the problem for all applications.

The level of parallelization possible could also be reduced by problems involving composition of multiple parallel libraries operating simultaneously.

On the other hand, writing parallel applications is very hard, and retrofitting concurrency on existing applications is even harder. Parallelizing compilers [1] try to automatically extract concurrency from a sequential program description, while still maintaining program correctness. Although these compilers have made significant advances, they still fail to parallelize many applications, because of data and interprocedural dependencies. A solution to this problem is speculative parallelization, running application code under a memory transaction that can be aborted if a memory access conflict is detected. Many hardware-supported speculative execution systems have been proposed by researchers [2–4]; this work proposes a new approach, using software transactional memory [5] to support the speculative execution.

Supported by hardware transactions, speculative parallel threading, also known as thread-level speculation (TLS) tries to parallelize applications at a very fine-grained level. A common approach is to generate transactions for a loop, executing each loop iteration inside its own transaction. Other approaches include grouping application code in “tasks” that are run in parallel. All these approaches rely on, and are limited by, the transactional support offered by the hardware. Transactions have to be very small, and as a result are very often spawned and
completed, and immediately squashed when they have potentially conflicting memory accesses. Also, for hardware transactions to work there has to be, obviously, transactional support offered by the hardware. Most current and upcoming hardware does not have this support, and most of the research results presented thus far are obtained with software simulations of this hardware. All these limitations are not new, and are some of the reasons underlying the proposal of software transactional memory.

Software transactional memories allow arbitrarily big transactions, and by using versioning, reduce the number of aborted transactions because of read conflicts. Also, one of the biggest advantages of STMs is that they work with current, past, and upcoming hardware, because they do not require low-level transactional support, and are usually implemented on top of common synchronization primitives like Load-Linked/Store-Conditional that are present in some form on most architectures.

This work proposes to explore speculative parallel execution of legacy single-threaded Java applications on a multicore architecture.

The main objective is the automatic transformation of a Java application, so that it will take advantage of multiple processing cores by speculatively running parts of the original sequential code concurrently in multiple STM transactions. The choice of Java applications is an implementation detail; the results of this work can also be mapped to other popular object-oriented programming languages.

Also of importance is an analysis of which parallelization techniques work and don’t work in this approach, and of bottlenecks caused by the underlying STMs and Java virtual machines. Furthermore, I hope to identify STM features that are important and penalizing for a speculative execution system.

In addition, because the field of speculative execution supported by a software transactional memory is not widely discussed in current research, I hope to provide insights on problems, solutions, and possible future research options on this field.

3 Related Work

The following sections describe important research related to this work in the fields of speculative execution, parallel programming and parallel implementations of libraries, hardware and software transactional memory, transactional execution and automatic parallelization. Section 3.7 concludes by discussing and expanding some of the ideas presented in the previous sections.

3.1 Speculative execution supported by hardware transactions

POSH [2] presents a thread-level speculation (TLS) infrastructure on top of the GNU Compiler Collection (GCC) that targets a multiprocessor architecture with hardware support for speculative transactions. The POSH framework is composed of a compiler and a profiler. The compiler has three main stages: task
selection, spawn hoisting, and task refinement. Task selection uses a variety of heuristics to identify different tasks, which can still have dependencies – the hardware ultimately guarantees correct execution; spawn hoisting tries to hoist the transaction spawn instruction as much as possible with some restrictions; and in the task refinement phase the final set of tasks is selected. The profiler runs applications with a train input set, and provides the compiler with a list of tasks that are beneficial for performance, allowing the compiler to eliminate non-beneficial tasks. The authors argue that even when a speculative execution aborts – when a task is \textit{squashed} – it may still be beneficial for the program execution, as when the task is restarted some of the values needed are already in cache, and so a squashed task also works as a prefetching mechanism. Benchmarking with SPECint 2000 on a simulated 4-core TLS multiprocessor shows an average speedup of 1.30.

In [3] the authors present a reverse compilation framework that translates binary code to static single assignment (SSA) form, from there performing optimizations and adding support for speculative execution. Profiling is also used to identify important program areas, and to predict likely values, reducing the number of speculative mis-predictions and their penalties. The approach of reverse compilation allows speculative parallelization without access to source, which is important to many legacy applications for which the original source code has since been lost or on which access to the source code is refused by the vendor.

Bridges et al. [4] describe a framework that combines whole-program analysis and speculative execution with extensions to the sequential programming model that further enable parallelization. The suggested extensions allow the programmer to specify that multiple legal outcomes of the program execution are possible:

- The \textbf{Y-branch} extension is similar to a common \texttt{if-then-else}, but it allows the true path to be taken regardless of the condition of the branch; the compiler is free to generate code to pursue this path when it is profitable to do so (Figure 1).
- The \textbf{Commutative} extension informs the compiler that calls to a function (or group of functions) can occur in any order; this is particularly useful for functions that have internal state – and as such generate dependencies – but on which the order of the calls are not relevant to the application as a whole, like the \texttt{random} function (Figure 2).

Chen et al. present Jrpm [6], a Java Virtual Machine that does TLS on a multiprocessor with hardware support, which coupled with a hardware profiler, analyzes speculative buffer requirements and inter-thread dependencies at runtime to identify loops to parallelize. Once sufficient data has been collected, the selected loops are dynamically recompiled. The use of Java bytecode allows the integration of the new execution model without modifying the source binaries, nor needing reverse compilation or source code.

Oplinger et al. [7] investigate the problem of how to find and exploit speculative thread-level parallelism. The authors found that it is inadequate to exploit
while (b = readByte()) {
    profitable = compress(b, dictionary);
    @YBranch(probability=.0001)
    if (!profitable)
        dictionary.restart();
}

Fig. 1. Example use of a Y-Branch on a simplified dictionary-based compression algorithm. The compiler is free to restart the dictionary, even if profitable is true.

private long seed;

@Commutative
protected synchronized int next(int bits) {
    seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
    return (int) (seed >>> (48 - bits));
}

Fig. 2. Implementation of the java.util.Random.next() method, that generates a pseudorandom number based on a seed, annotated with the @Commutative extension.

only loop-level parallelism, as many systems do, and that procedure calls provide important opportunities for parallelism, so they propose the use of speculative procedure execution, coupled with procedure return value prediction. The presented results were obtained by simulating the execution of applications on variations of an optimal speculative thread-level parallelism (STP) machine; it is argued that this optimal machine can be used to derive an upper bound on the performance achievable on any real machine of a similar design. Speculative procedure execution is done by concurrently executing a called procedure with the code following the return of the procedure. This approach is then applied recursively. As it is common that the return value of a procedure is immediately used, this is where the proposed procedure return value prediction is applied. This prediction works by keeping a cache of past values returned by a procedure, that are then used to speculatively run the code following the procedure return immediately; in case of mis-speculation, the speculative thread is aborted, and the code re-run with the real value.

3.2 Parallelization of common shared libraries

Parallel math algorithms [8] have a long history of research; parallel math libraries are commonly used for speeding up scientific applications on multipro-
cessor machines. For general (non-scientific) libraries, not much work has been published on parallel implementations.

The MCSTL [9] is a parallel implementation of the C++ Standard Template Library (STL). The STL is part of the C++ Standard Library, and as such, used by most nontrivial C++ applications. By using a parallel implementation of the STL, these applications can experience a speed up when running on a multicore architecture. The work on MCSTL demonstrates the validity of speeding up applications by parallelizing the libraries they depend on, but also shows that an efficient parallel implementation of a common library is a very complex task.

3.3 Hardware Transactional Memory

Transactional Memory [10] was initially proposed by Herlihy and Moss as a multiprocessor architecture capable of making lock-free synchronization as efficient and easy to use as conventional techniques based on mutual exclusion, while avoiding pitfalls like priority inversion, convoys, and deadlocks. The implementation was based on extensions to multiprocessor cache-coherence protocols, adding some new instructions to the processor, and a small transactional cache where transactional changes were kept prior to committing. It was a direct generalization of Load-Linked/Store-Conditional, providing the same semantics for multiple memory words, instead of being restricted to just one. But, it was not dynamic: memory usage and the transactions had to be statically defined in advance.

Software transactional memories have since been proposed that overcome many of these problems, at the cost of bigger overheads.

3.4 Software Transactional Memory

Software transactional memory [5] was introduced as an alternative to hardware transactional memory that could be implemented on top of Load-Linked/Store-Conditional of a single word, as provided by most current hardware architectures. The authors recognized that although hardware transactional memory was a very promising tool for implementation of non-blocking concurrent programs, applications using it were not portable, and most architectures being introduced and planned at that time (and today still) did not incorporate HTM into their designs. As it was the case with Herlihy’s original HTM, this STM was limited to static transactions, where the data set is known in advance, such as a k-word compare-and-swap.

The Dynamic Software Transactional Memory (DSTM) [11] was the first unbounded STM, allowing it to be used in the implementation of dynamically-sized data structures such as lists and trees. It also allows transactions to detect if they will cause some other transaction to abort before doing so, therefore allowing the decision of whether to proceed or to give the other transaction a chance to complete its work first. Such decisions are made by pluggable contention managers.
The McRT-STM [12] is an STM implementation that is part of McRT, an experimental Multi-core RunTime. This STM supports advanced features such as nested transactions with partial aborts, object based conflict detection for C/C++ applications, and contention managers. The authors also present an analysis of STM design tradeoffs such as optimistic versus pessimistic concurrency, write buffering versus undo logging, and cache line based versus object based conflict detection. Unlike many designs, the McRT-STM does not provide non-blocking guarantees, because the authors argue that the implementation is both simplified and more efficient this way. Memory changes are done in-place on writes; the location’s original value is saved on an undo log, that is used in case the transaction aborts.

The Java Versioned Software Transactional Memory (JVSTM) [13–15] is a pure Java STM that introduces the concept of versioned boxes. Versioned boxes are containers that keep the history of the values of an object, each of these corresponding to a change made to the box by a committed transaction. This arrangement allows read-only transactions to never conflict with any other concurrent transaction, and favors applications that have a high read/write transaction ratio. The authors recognize that read-only and read-write transactions present different implementation overheads, and the importance of distinguishing between them. Also proposed are speculative read-only transactions, where the JVSTM speculatively assumes that a transaction is read-only when it starts, and where the transaction is aborted and restarted as a read-write transaction if it tries to change a versioned box; and garbage collection of old history values that are not needed anymore, because all transactions that could reach them have either committed or been permanently aborted. The JVSTM also supports medium (using hundreds to thousands of java objects) and long-running transactions (using thousands to millions of objects).

3.5 Hybrid Transactional Memory

Hybrid Transactional Memory (HyTM) [16] is an approach to implementing a transactional memory in software that can use best-effort hardware support to boost performance, but does not depend on it. As such it works on all computers, with or without hardware support for transactions. When hardware support is available, transactions try to run with this support; if they reach hardware limitations they abort and restart as software transactions. The system is implemented as a prototype compiler based on the Sun Studio C/C++ Compiler and a STM library; the compiler produces code for executing transactions using HTM or using the STM library. An important part of this work is reconciling concurrent hardware and software transactions, especially detection of conflicts between them. Additionally, this hybrid approach allows for an incremental and transparent transition between pure software TM, and hardware TM, allowing chip designers to gradually introduce transactional support in their chips, without having to commit to an unbounded (dynamic) hardware solution from the start. The authors also recognized the importance of the contention managers proposed by Herlihy et al. [11].
Rajwar, Herlihy and Lai [17] present the Virtual Transactional Memory (VTM), that like the HyTM, combines hardware and software approaches. The authors argue that like virtual memory shields the programmer from platform-specific limits of physical memory, so must virtual transactional memory shield programmers from HTM limitations such as transactional buffer sizes, scheduling quanta and page faults. But, unlike the HyTM, VTM also needs hardware machinery to handle the transition from working in the “hardware-only” mode (HTM) to the unbounded transaction mode.

3.6 Transactional Java Execution

Carlstrom et al. [18] investigate the implications of using hardware transactional support to execute existing parallel Java applications. The general approach proposed is the transformation of synchronized blocks into atomic transactions. The authors argue that strong transactional atomicity semantics are a natural replacement for the critical sections defined by synchronized. Also discussed are the problems underlying calls to native machine code through the Java Native Interface (JNI), and of non-transactional operations. The authors conclude that existing parallel Java applications can be run transactionally with minimal changes, and that a continuous transactional system can deliver performance equal to or better than the pessimistic lock-based implementation of the application.

In [19], the authors present the Atomos programming language, the first programming language with implicit transactions, strong atomicity, and a scalable multiprocessor implementation. Atomos is derived from Java, replacing Java synchronization and conditional waiting with transactional alternatives. The current implementation is based on the Jikes RVM [20] Java virtual machine, and on the Transactional Coherence and Consistency (TCC) hardware transactional memory model [21]. In Atomos, transactions are defined by an atomic statement that conceptually replaces the use of synchronized statements. The watch statement allows programmers to specify fine-grained watch sets, that are used with the retry conditional waiting statement for transactional conflict-driven wakeup. Also included are open-nested transactions, using the open statement, which are nested transactions that can be committed and their results seen by other transactions, even while the parent transaction is still active; this allows threads to communicate between transactions, similar to volatile variables, while minimizing the risk of data dependency violations. Transaction commit and abort handlers that run on transaction commit or abort are supported, along with violation handlers that allow programs to recover from data dependency violations without rolling back. Benchmarks were done on a PowerPC CMP system simulator with support for the TCC HTM, and show a clear advantage of Atomos over Java for the tested applications.
3.7 Discussion

Following is a discussion of some issues and proposals made in the previous sections.

3.7.1 Annotations

The use of annotations to add metadata that guides parallelization, similar to [4] is a very interesting approach. Annotations like the proposed @Commutative help to break interprocedural dependencies, decreasing the number of aborted speculative transactions.

Other interesting annotations could inform the parallelization system of variables and sections of the code that do not need to be kept transactional, and of code zones that have high contention, and as such are poor candidates for speculative execution.

Unfortunately, this approach is error-prone, as by deviating from the original sequential program state, bugs may be introduced (such as erroneously marking a procedure with @Commutative), and it needs to be done by a programmer with sufficient knowledge of the program codebase.

3.7.2 STM Features

Many STM models have been proposed, representing different design choices that lead to different overheads in transaction start and commit, conflict detection, conflict avoidance, contention management, reading and writing to memory positions, and transaction size and duration limits; features such as support for restartable nested transactions and best-effort hardware transactions; and different non-blocking progress conditions.

A speculative parallelization system potentially has a very different use of STM features than most applications that explicitly use STM. If the underlying STM needs to add indirections for accesses to program memory, the additional performance obtained from the speculative execution needs to overcome this extra source of overhead to get a better performance than the original sequential code.

Also, STM features that benefit read-only transactions and tend to penalize read-write transactions might be counter-productive for many desktop applications. The rate at which transactions can be spawned and committed, as well as the overhead of their creation, also limit most very-small-grained speculative executions.

3.7.3 Speculative Execution Backed by HTM vs Backed by STM

Because software transactional memory presents different, bigger, overheads when compared to most hardware transactional memory implementations, the approaches taken on STM-backed speculative execution cannot be the same as those proposed by current HTM-backed speculative execution research. The risks of misprediction are also higher: the longer a speculative transaction is running,
the bigger the probability that it will not complete successfully, and the bigger
the potential waste of resources.

As such, new approaches may have to be developed to take advantage of
medium-grained transactions.

4 Architecture

The system that I propose to develop during the course of this work will analyze
and make changes to Java applications, so that parts of the code are automatic-
cally and speculatively run in multiple transactions, in parallel.

As it was already discussed, if a speculative transaction fails, no correcting
measures have to be taken, the transaction just has to be restarted until it
commits successfully.

The system will be composed of two main elements:

- Static Analysis and Parallelization Module – This module will be responsible
for the identification of speculative transaction spawn and commit points,
and for modifying the original application to run with a STM, spawning and
committing at those points (Figure 3).
- Runtime Control Module – This module will be able to gather runtime statis-
tics, to try and reduce the rate of mis-speculated transactions; it will also
contain some runtime interfaces used by the inserted speculation code (Fig-
ure 4).

An important part of this work will be the analysis of which parts of an
application are parallelizable, and which parts are not, along with determining
in which of those cases it is profitable to apply parallelization.

Two promising approaches for this parallelization are concurrent execution
of method (procedure) calls and the code that follows them (Figure 5); and spec-
ulative loop parallelization. For the first case, care has to be taken to not choose
very small procedures, which are common in current object-oriented program-
ing languages, including Java. Also, return value prediction as proposed by [7]
might help for the cases where the return value for a method call is immediately used after the call. For speculative loop parallelization to yield speedups, even in the face of STM overheads, multiple loop iterations could be run in a single speculative transaction, resulting in a scheme similar to loop unrolling.

The planned system will not depend on a specific STM design/implementation, to allow experimentation with multiple designs proposed in current research.
void methodA() {
    methodB();
    methodC();
    return methodD();
}

Fig. 6. Example code for methodA.

4.1 Java Language and Virtual Machine

The Java language [22] and virtual machine [23] have the goal of providing a portable, object oriented, garbage collected, high performance environment for applications to execute.

Java applications are typically compiled to a low-level standard bytecode representation that can be run on any Java Virtual Machine (JVM). This standard bytecode representation makes it straightforward to analyze and modify current applications; it also allows custom virtual machine implementations to seamlessly integrate support for hardware and software transactions, while still preserving the original sequential code semantics [24].

The Java memory model is also clearly defined [25], providing clear semantics for the legal behaviors of parallel applications like STM libraries. This way both the original java applications and their speculative execution versions can be run on all architectures that feature a compliant virtual machine.

4.2 Implementation Concerns

The following sections discuss some concerns that a speculative execution system has to take into account.

4.2.1 The Irreversible Side Effects Problem

Many STM designs suffer from the same problem as functional programming languages: how to deal with side effects. A function or expression is said to produce a side effect if it modifies some state in addition to returning a value. Irreversible side effects typically occur in calls to native code (for example the underlying OS or virtual machine) for things like I/O, where it is generally not possible to undo the effect of the call in the case of an aborted transaction.

STM systems normally deal with this problem by forbidding these kinds of operations inside transactions that may be aborted. An alternative solution to the I/O problem could be the use of buffers everywhere, but this solution is not definitive; as an example consider an echo client/server application: if the client

\footnote{Taken from: http://en.wikipedia.org/wiki/Side_effect_(computer_science)}
tries to read a value from a TCP/IP socket that is the result of a former write to that socket inside the same transaction, as the write is buffered until the end of the transaction, no response will ever appear, because only when (and if) the transaction is committed will the information be transmitted to the remote host, and will the remote host supply the local client with an answer.

One way a speculative parallel execution system may solve this problem, is by identifying which sections of the code produce irreversible side-effects, and by making sure that they only execute once, and if present inside a transaction, that the transaction will never abort.

4.2.2 Speculative Parallelization of Already Parallel Applications

Trying to speculatively parallelize applications that already make use of multiple threads raises some issues. It might be difficult to determine if an application will run using multiple threads, because these threads might be created by external libraries and not by the application itself. Pessimistic thread synchronization constructs might have to be replaced with atomic transactions. Threads that are started inside a transactional context also pose problems on how they should be handled by the underlying STM, or if they should be supported at all.

Because the same benefits of automatic speculative sequential application parallelization might apply to these applications also, this approach should not be overlooked. Many parallel applications use a very coarse-grained parallelization scheme, so they could potentially benefit from further fine-grained parallelization. The substitution of pessimistic locking with optimistic transactional execution may also benefit existing parallel applications.

5 Work Evaluation

This work will be tested by timing several sequential java applications on a multicore machine, versus their original sequential runtime. Multiple software transactional memory models will be tested, with the goal of identifying important characteristics these models need to provide for successful and performant speculative execution.

The identification of applications to be tested is a future work.

The results of this work will be important even if most applications do not show performance improvements: if the obtained results are not favorable, I hope to identify why this happens, specifically how different STMs and Java virtual machines impact these results, and to document programming patterns that are favorable and unfavorable for this approach.

6 Conclusion

Parallel programming is hard. Because of that, most multicore machines today do not realize their full potential, as most applications are not prepared to take advantage of multiple processing cores.
Additionally, newer chips present little advancements to sequential execution speed, instead incorporating more processing cores, bringing speedups only to parallel workloads.

Successfully and efficiently retrofitting an existing application to take advantage of multicore machines is even harder.

This work proposes to explore speculative parallelization of existing sequential Java applications, supported by software transactional memory. The goal is to allow legacy applications to automatically take advantage of multicores, by speculating that certain parts of the program can be run in parallel, instead of in sequence. If anything goes wrong (deviates from the original sequential program state), the underlying software transactional memory detects this as a conflict between transactions, and aborts one of the transactions; the application state maintaining its correctness.

7 Acknowledgements

This work is being performed in the scope of the Pastramy project (PTDC/EIA/72405/2006), which is funded by the Portuguese FCT (Fundação para a Ciência e a Tecnologia).

References

25. Manson, J., Pugh, W., Adve, S.: The Java Memory Model