JaSPEx: Speculative Parallelization on the Java Platform*

Extended Abstract

Ivo Anjo
Instituto Superior Técnico/Universidade Técnica de Lisboa

Abstract. Multicore processors, capable of running multiple hardware threads concurrently, 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 into more sequential performance, and existing sequential applications will not speed up by moving to a multicore. To tackle this problem, I propose to use thread-level speculation based on a Software Transactional Memory to parallelize automatically sequential programs. I describe the JaSPEx system, which is able to do automatic parallelization of existing sequential programs that execute on the Java Virtual Machine. I address the problem of transactifying an existing program and the difficulties inherent to this process, and I describe how speculation is introduced and controlled by the JaSPEx system.

Key words: Speculative Execution, Thread-level Speculation, Software 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, 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.

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. Thus, an enticing alternative is to parallelize applications automatically. In fact, there is already significant research towards this goal.

For instance, parallelizing compilers [1] try to automatically extract concurrency from a sequential program description, while still maintaining program

* This work was partially funded by an FCT research grant in the context of the Pastramy project (PTDC/EIA/72405/2006).
correctness. The problem is that they still fail to parallelize many applications, because of data and interprocedural dependencies that are very hard to analyze at compile-time in a fully static way.

This work explores a different approach – speculative parallelization. Rather than parallelizing only code that is provably able to run in parallel, speculative parallelization uses a more aggressive approach that parallelizes code that may have dependencies, and relies on the ability to roll back a speculative execution when it detects that the parallelization could not have been done.

Unlike other approaches to automatic parallelization that rely on hardware-supported speculative execution (e.g., [2,3]), the distinguishing feature of my proposal is the use of a software transactional memory (STM) [4] to back up the speculative execution. To the best of my knowledge, I am the first to propose the use of an STM for speculative parallelization.

I argue that using an STM for speculative execution has several advantages over hardware-supported approaches. First, because STM-based executions are unbounded, I may extend the range of possible speculative parallelizations, thereby increasing the potential for extracting parallelism from sequential applications. Second, I may apply these techniques to applications that run on hardware that does not support speculative execution (including all of the current mainstream hardware). Finally, I may leverage on much of the intense research being done in the area of transactional memory.

In this paper, I describe JaSPEx – the Java Speculative Parallel Executor – a system that automatically parallelizes programs for the Java Virtual Machine (JVM) using an STM-based speculative approach. JaSPEx rewrites the bytecode as it is loaded by the JVM runtime, modifying it to run speculatively.

The remainder of this work is organized as follows. Section 2 introduces problems and solutions found for running code speculatively. Section 3 presents experimental results. Section 4 presents important research related to this work, and, finally, Section 5 summarizes the current findings and future work.

2 Design and implementation

We may parallelize the execution of a Java method like the one shown in Figure 1 by executing the calls to doA and doB in parallel. The problem is, these methods might modify and access some shared state, and as such may not be able to run in parallel.

```java
void method() {
    doA();
    doB();
}
```

Fig. 1. Example method to be parallelized.
Using a speculative approach to the parallelization of programs entails having the ability to detect when a speculative execution violates sequential execution semantics, and the ability to reverse the changes done by a speculative execution when such a violation occurs.

JaSPEx consists of two main parts: (1) transactification, that modifies an application to run under the control of a STM; and (2) speculative execution, that further modifies applications and performs the speculative executions, coordinating the start, end, termination and return of values from these executions.

Applications are modified at load-time, via Java bytecode rewriting, using the ASM bytecode manipulation framework [5]. Because looking into the transformations made at the bytecode level makes them harder to understand, I present the transformations as semantically equivalent changes in Java.

2.1 Transactification of an application

Because the JVM runtime has no support for transactional execution of code, an application must first be modified to run transactionally, so that the automatic parallelization system is able to detect when a speculative execution violates sequential execution semantics, and is able to reverse the changes made by a speculative execution when such a violation occurs.

To solve this problem, I propose the use of a software transactional memory [4] to allow (parts of) the program memory to act transactionally. Execution of different parts of the application is then mapped to different transactions each executing on their own thread, and when there is a conflict between two transactions we know that there has been a violation of sequential execution semantics, and abort the one that comes later in the original program execution.

Coming back to the example in Figure 1, we can parallelize execution of method by running doA and doB in separate threads, each with a different transaction. If the STM system detects a conflict between the speculative execution of doA and doB, we abort doB, and schedule it for reexecution, because the original program order puts doA before doB; if no conflict is detected, the two methods are run in parallel, and this should result in a speedup over the sequential version.

The software transactional memory currently used for JaSPEx is the Java Versioned Software Transactional Memory (JVSTM) [6,7], which is a pure Java STM that introduces the concept of versioned boxes, which 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. The JVSTM was chosen for its features and due to my familiarity with it, but my approach can also be used with other STMs.

As a Java library, applications have to explicitly call the JVSTM to start and end transactions, and Java classes have to be modified to hold jvstm.VBox instances, instead of instances of the original object types, as shown in Figures 2 and 3. This process, which I call transactification of a class, has to be applied to all classes of a target application, so that it runs entirely under the control of the JVSTM.

The transactification process does the following:
public class A {
    private String s;
    public A(String s) { this.s = s; }
    public String s() { return s; }
}

Fig. 2. Original A class.

public class A {
    private VBox<String> $box_s = new VBox<String>();
    public A(String s) { $box_s.put(s); }
    public String s() { return $box_s.get(); }
    private String $box_s_get() { return $box_s.get(); }
    private void $box_s_put(String s) { $box_s.put(s); }
}

Fig. 3. Transactified A class.

- Replaces the original fields of each class with private VBox<OriginalType> fields named $box.FieldName.
- Creates the get and put methods, $box.FieldName.get and $box.FieldName.put, which mediate access to the corresponding VBox. These methods have the same access level as the original field.
- Adds VBox slot initializations to the class constructors.
- Replaces accesses to the original fields, either from the same class or from outside classes, with calls to the get and put methods.

Unfortunately, not all things can be transactified. For instance, native methods cannot be analyzed or modified easily. Also, the Sun JVM reserves the java.* package namespace and does not allow loading at runtime modified versions of classes within this package or any of its subpackages. I refer to a class that cannot or should not be modified as an unmodifiable class.

Besides these unmodifiable classes, there are other features of the Java language and runtime that make the transactification process harder. Arrays cause a multitude of problems. Not only because individual array positions have to be transactified, but specially because transactifying them causes changes to the API of transactified classes, as arrays of a given OriginalType have to be replaced by arrays of VBox instances. This means that all method signatures receiving or returning arrays have to be changed to accommodate this change, which, as I stated before, is not possible on the Sun JVM. Another source of problems is the use of reflection, because it eludes the static transformation of accesses to fields. So, reflection has to be forbidden during speculation, or else modified to be speculation aware. Finally, I/O operations generally cannot be undone.

Because not all things can be transactified, JaSPEx must be able to detect all of these cases and make sure such invocations are forbidden during speculative execution.
2.2 Prevention of nontransactional operations

There are two main approaches to prevent the execution of nontransactional operations within a speculative execution: (1) static identification of these operations; and (2) dynamic, runtime prevention of their execution. Static identification consists of building a graph of possible method invocations: If method $A$ may call native method $B$, then both $A$ and $B$ are marked as nontransactional. Note that, even though there may be a control flow from $A$ to $B$, it does not mean that $A$ calls $B$ each time it executes. So, as this approach is very conservative, I opted for a dynamic runtime scheme, where methods are modified to invoke the speculation system to check if they can perform nontransactional operations.

To support speculative execution, JaSPEx creates a speculative version of each method $M$, called $M_{\text{speculative}}$, except for constructors, which always have to be named $\text{<init>}$. In this latter case, an alternative scheme is used: A new parameter of type $\text{SpeculativeCtorMarker}$ is added at the end of every speculative constructor. The speculative version of each method is a copy of the original method with invocations to other methods replaced by calls to their $\text{speculative}$ versions, if possible; for nontransactional method invocations, nontransactional field accesses, and operations involving arrays, it adds an invocation to the JaSPEx runtime before performing the operation, so that the speculation system can decide how to proceed.

Additionally, if the original method was native, its $\text{speculative}$ counterpart consists of a call to the JaSPEx runtime, followed by a call to the original version. Similarly, because a class may inherit methods from an unmodifiable class, it needs to add $\text{speculative}$ versions of inherited methods that call the runtime and then the original method on the superclass.

Figures 4 and 5 exemplify the application of some of these changes.

2.3 Doing speculation

After the transactification and the addition of support for handling nontransactional operations, a final round of modifications for speculation is introduced. These allow the speculation system to know when it can spawn a speculative execution, and when the speculation results should be applied or discarded.

Currently, JaSPEx speculates only on method executions: When a method is invoked, some of the methods it invokes may be run speculatively. Speculation is only considered for methods in which their arguments can either be determined statically or are simple dynamic cases like arithmetic operations or parent method argument accesses. As an example, consider a recursive implementation of the Fibonacci function shown in Figure 6.

At each call to $\text{fib}$, JaSPEx speculatively launches the execution of $\text{fib}(n-1)$ and $\text{fib}(n-2)$ and then proceeds with the execution of the method: In the case where $n \leq 1$, the speculative executions that may be running are discarded;

---

1 I consider accesses to unmodifiable classes to be nontransactional, including their fields.
public class B {
    public B() { System.out.println(toString()); }
    public String toString() { ... }
}

Fig. 4. Transactified B class.

public class B {
    public B() { ... } // Same as original
    public String toString() { ... } // Same as original
    public B(SpeculativeCtorMarker marker) {
        SpeculationControl.nonTransactionalActionAttempted(...);
        System.out.println(toString$speculative());
    }
    public String toString$speculative() { ... }
}

Fig. 5. B class after introduction of $speculative versions of methods.

java.lang.System is an unmodifiable class, so access to its field `out` is considered a nontransactional action, as is the invocation of `println`.

```java
public static int fib(int n) {
    if (n <= 1) return n;
    return fib(n-1) + fib(n-2);
}
```

Fig. 6. Fibonacci function.

otherwise, their results are retrieved and the transactions that they are running in are committed, if possible.

Speculative methods call the JaSPEx runtime when they are started, before they terminate, and to get results from speculative executions. When a method starts, it calls the method `SpeculationControl.entryPointReached`, passing as arguments an entry-point id that uniquely identifies each speculative method, and an array of arrays with the arguments for each function call that is to be executed speculatively within that method. For instance, in the `fib` example, we will execute speculatively `fib(n-1)` and `fib(n-2)`. Thus, `entryPointReached` will receive an array `arr` of type `Object[2][]`, where `arr[0]` contains `n-1`, and `arr[1]` contains `n-2`. As a result of the call to `entryPointReached`, an instance of `SpeculationId` is returned, which identifies the current dynamic execution context uniquely.

Before a method exits, a call to `SpeculationControl.exitPointReached` is made, to inform the runtime that the method will terminate, and that speculative executions that might be queued or running for this method should be discarded.
public int fib$speculative(int n) {
    SpeculationId specId =
      SpeculationControl.entryPointReached(ENTRY_POINT_ID,
              new Object[] { new Object[] { n-1 },
                            new Object[] { n-2 } });
    if (n <= 1) {
       SpeculationControl.exitPointReached(specId);
       return n;
    }
    Future f0 = SpeculationControl.getResult(specId, INV_ID_0);
    Future f1 = SpeculationControl.getResult(specId, INV_ID_1);
    int temp = f0.get() + f1.get();
    SpeculationControl.exitPointReached(specId);
    return temp;
}

Fig. 7. The speculative version of the Fibonacci function. The symbols INV_ID_0*
identify the function calls that they replace: In this case, INV_ID_0 represents
the call to fib(n-1), whereas INV_ID_1 represents the call to fib(n-2).

Method invocations for methods that are executed speculatively are replaced
by calls to SpeculationControl.getResult, which, given the current Specu-
lationId and an identifier that identifies the function call, returns a Future
object that represents the result of the speculative execution. Finally, to obtain
the result, get() is called on the Future; if the underlying method execution resulted
in an exception being thrown, that exception will be rethrown by get().

Figure 7 shows the fib$speculative method with these modifications.

2.4 Runtime control

As seen in the previous sections, calls to methods of the class SpeculationControl
are added at various points of the speculative methods.

A speculation starts with a call to SpeculationControl.entryPointReached
which, as seen before, receives an entry-point id and an object array containing
arguments to be used for speculative calls. The entry-point id is used to access
a list of instances of java.lang.reflect.Method, each of which corresponds to a
method that is going to be executed speculatively. JaSPEx generates a new task
for each element in this list: Each task will compute a call to the correspond-
ing method with the appropriate arguments. For instance, for the execution of
fib(n), two smaller tasks are generated, representing the calls to fib(n-1) and
fib(n-2). These tasks are queued for execution by worker threads.

When a worker thread picks up a task, it starts a new STM transaction
and uses reflection to invoke the method with the supplied arguments. Because
the method executes within an STM transaction, none of its changes are visible
to the outside until the transaction commits. Moreover, if, during the method
execution, it tries to execute a nontransactional action, it stops and waits for permission to commit its current STM transaction – it waits until it can run on normal, sequential program order, so that it cannot be aborted. If, instead, the method terminates with a return value or an exception, it also waits for permission to be committed. Finally, as a method running speculatively may also cause other speculative execution tasks to be created, when a method wants to give permission to commit to a method speculation that it started, it also has to wait for permission to commit its own transaction first.

Permission for a speculative task to commit is given only when the method get is called on the Future representing the task (itself a result from a call to the method SpeculationControl.getResult) and that call is made by the thread currently running in the normal program order. This scheme results in speculative transactions being committed in original, sequential program order, as expected. If a conflict is detected when trying to commit a transaction, the task is aborted and reexecuted; this time, it will commit for sure, because it is executing in the original program order.

3 Experimental results

I now present some preliminary results of the automatic parallelization performed by JaSPEx. These results were obtained on a dual-quadcore system with two Intel Xeon E5520 processors, running Ubuntu Linux 9.04, and Java 1.6.0.13.

As fib does very little computation at each step, I have modified it to do speculative execution only up to a threshold, and from then on to run the rest of the computation entirely without speculative execution on the same thread. Figure 3 presents the time needed for calculating fib(50).

These are very preliminary results, but they are encouraging, showing that it is possible to automatically extract parallelism from a sequential program with the approach that I propose.

![Fig. 8. Time for calculating fib(50) using speculative parallelization, as the number of available cores is increased.](image-url)
4 Related work

Transactional Memory was initially proposed by Herlihy and Moss [8] as a multi-processor architecture capable of making lock-free synchronization as efficient and easy to use as conventional techniques based on mutual exclusion.

Software transactional memory was later introduced as an alternative to hardware transactional memory [4] that could be implemented using Load-Linked/Store-Conditional of a single memory word, as provided by most current hardware architectures.

Many hardware-supported thread-level speculation (TLS) systems have been proposed by researchers. POSH [2] presents a TLS infrastructure on top of the GNU Compiler Collection, composed of a compiler and a profiler; it parallelizes applications by analyzing the source code and using heuristics to identify tasks, which can be further refined by using the profiler. Jrpm [3], the Java runtime parallelizing machine is a Java virtual machine that does TLS on a multiprocessor with hardware support. It analyzes buffer requirements and inter-thread dependencies at runtime to identify loops to parallelize, and dynamically recompiles the selected loops.

The primary difference between these systems and JaSPEEx is my use of a software-based TM. Because a software-based TM has no inherent limits to transaction duration and size, I expect to be able to extract more parallelism from an application, that is available only at a higher level.

FJTask [9] is a framework that supports fork/join parallelism, a style of parallel programming where problems are solved by recursively splitting them into subtasks, which can then be executed in parallel. Welc et al. [10] introduce safe futures for Java, which are futures that work as semantically transparent annotations on methods, where execution of a method can be replaced for execution of a future, but where sequential execution semantics are respected.

My current implementation is very similar to the fork/join style of programming: Speculative tasks are created at the beginning of $speculative$ methods, and the joins are done at the original method call sites. Unlike FJTask [9], though, where algorithms need to be explicitly modified to use fork/join calls, my framework tries to do a similar conversion automatically. The work of Welc et al. [10] is also similar to mine, because it allows multiple parts of the code to run speculatively in parallel. Unlike mine, however, the program needs to be manually modified to use the safe futures, and depends on their modified JVM for execution.

5 Conclusions and future work

In this paper I proposed to use an STM-based approach to thread-level speculation, so that I may extract more parallelism from sequential programs, benefit from the results of the transactional memory research community, and target current hardware.
I have incorporated my proposal into a running system – JaSPEx – that automatically parallelizes a program that was compiled to run in the Java Virtual Machine. To accomplish that, JaSPEx transforms the program, without the intervention of the programmer, so that some parts of it may execute speculatively. One of the challenges in this transformation is the transactification of the program. In this work I describe some of the difficulties inherent to the transactification of a JVM program if we have no support from the JVM runtime.

In its current state, JaSPEx shows promising results – obtaining linear speedup on a recursive implementation of the fibonacci function – even though it has not been tested on realistic benchmarks, yet. Nevertheless, in the future I hope to obtain further speedups by reducing overheads in task creation, commit, and abort; by implementing a more dynamic system that gathers statistics on speculation duration and success rates, with the objective of avoiding method speculations for very small methods and for methods with high abort rates; and by further optimization of the JVSTM for my use-case, thereby reducing overheads in transactional execution of applications.

References