Task Scheduling in Speculative Parallelization

David Baptista
Instituto Superior Técnico
Universidade Técnica de Lisboa
Av. Rovisco Pais, 1
1049-001 Lisboa
Portugal
david.baptista@ist.utl.pt

ABSTRACT
Concurrent programming using transactional memory models is an alternative approach to the ubiquitously supported lock-based programming. While hardware-based transactional memory has yet to become widely available, the advent of efficient, general purpose software-based transactional memory frameworks has made it possible to apply this approach to automatic, rather than manual, parallelization. There are still a number of challenges that need to be addressed, however, concerning decomposition of sequential programs into tasks, scheduling of said tasks, and conflict resolution, among others. In this paper, I address the challenge of scheduling parallel tasks, using as input programs parallelized by Anjo’s Jaspex system \[2, 1\]. I show how the concept of contention management, a concept borrowed from software transactional memory, can be merged with scheduling algorithms in order to obtain more efficient schedules. I also apply this principle to a general purpose transactional memory benchmark, STMBench7 \[9\], thereby showing how conflict-aware scheduling may enhance the performance of generic transactional memory frameworks.

Keywords
Transactional Memory, Speculative Execution, Automatic Parallelization, Multicore Environments, Contention Management, Scheduling

1. INTRODUCTION
Legacy applications are often written in a sequential fashion, garnering therefore no benefit from running on multicore architectures. On the other hand, even new applications are considerably more costly to develop as parallel applications. Therefore, the development and study of tools and frameworks to support development of parallel applications and parallelization of existing applications is a critical step towards maximizing the potential of multicores.

The goal of automatic parallelization is precisely to remove from the programmer the burden of having to deal explicitly with parallelism. This can be achieved in a myriad of ways, which fall into two categories: static and dynamic schemes. The first category consists of schemes that rely on offline program analysis to determine data and functional parallelism in the program, and modify it to run in parallel. On the other hand, dynamic schemes make decisions at runtime, and can take into account runtime information and properties. Systems in the latter category typically use a speculative approach to parallelization, assuming that a set of tasks can be run in parallel and performing some sort of rollback or cancellation upon stumbling on conflicts.

Software transactional memory is a maturing technology that has now reached the point where it is performant enough to be able to support both fine-grained and coarse-grained parallelism; two examples of state-of-the art performant software transactional memory runtimes are \[5, 6\] and \[12\]. Therefore, I believe the opportunity has surfaced to use software transactional memory as support for automatic parallelization systems, and some preliminary results on this possibility have been presented in \[2, 1\]. This approach to automatic parallelization seems promising, but there are also plenty of challenges to be addressed. One of these challenges is the scheduling of speculative tasks; the existing literature acknowledges, sometimes implicitly and often explicitly, the need of speculation-aware scheduling algorithms, rather than general-purpose schedulers, to take full advantage of speculative parallelization. In this paper, drawing on research from contention management, I implement a conflict-aware scheduler - a scheduler that uses information about past conflicts between tasks to iteratively produce more efficient schedules - on Anjo’s Jaspex \[2, 1\], an automatic parallelizer, and on the STMBench7, a generic transactional memory benchmark \[9\].

This paper is organized as follows: Section 2 describes what a speculative parallelization system is and how it works in generic terms, and why it is a good idea to use software transactional memory as speculation support. In Section 3 I describe what conflict-aware scheduling, and how I implemented it in both Anjo’s Jaspex system and in the STMBench7 benchmark. Section 4 presents the experimental results, and a discussion of their meaning. Section 5

*This work was partially funded by an FCT grant in the scope of the project RuLAM: Running Legacy Applications on Multicores, PTDC/EIA-EIA/108240/2008.
contains a review of important related work, and Section 6 concludes with my final thoughts and future directions for research.

2. SPECULATIVE PARALLELIZATION SYSTEMS

Automatic parallelization systems allow the creation of parallel programs from sequential programs, without intervention from the programmer. Speculative parallelization systems are a specific type of automatic parallelization systems, relying on an optimistic approach to parallelization: Two sections of code may be put into execution even if their parallel execution violates their sequential semantics, since the code is executed speculatively. This means essentially that the execution of a section of code can be revoked, and later re-executed.

Speculative parallelization systems can be generally divided into three main components. The first is the task identification component, which takes the sequential program as an input and identifies the sections of code (tasks) that are able to be autonomously executed by a single thread. The second component is the task spawning component. It accompanies the execution of the program, and is responsible for deciding whether it is or is not profitable to speculate at a certain point of the execution of the program, and for collecting the necessary data and context to package autonomous speculative tasks for execution. The third component is the speculation support component, which provides support for task speculation. As I review on Section 5, this third component is often a specialized speculation support scheme, which supports speculative execution of some kind of constructs (loop iterations, functions), but not others.

It is profitable to evolve towards a more generic support for speculation, and software transactional memory [15] seems to be a good solution. The programming model of software transactional memory allows the specification of sections of code of arbitrary size and complexity that may be executed and then revoked (the transaction aborted). Therefore, it is possible, at least in principle, to exploit parallelism at different granularities (statement, method, algorithm, and others). Hinging on this concept, Anjo developed a speculative parallelization system [2, 1], built on top of a software transactional memory framework, the Java Versioned Software Transactional Memory (JVSTM) [5, 6].

3. CONFLICT-AWARE SCHEDULING

The model of speculative parallelization systems I described above makes a potential hazard visible: in general, speculative parallelization systems do not have any control over how speculative tasks actually get assigned to threads. In practice, this means that this decision is often left to the operating system. Unfortunately, the operating system, having no information on the nature of these tasks, will only by chance produce a good schedule for these tasks.

Within the community of software transactional memory, a similar problem has been identified by Herlihy et al. [11], namely that of contention management. Although contention management in its seminal paper was more concerned with livelock and starving issues (that do not exist in the same form in the world of speculative parallelization systems), Herlihy et al. recognized how ignoring conflict patterns among competing transactions could severely degrade performance. The solution they developed, contention managers, creates an entity responsible for arbitrating conflict resolution between transactions (for instance, who gets aborted).

This concept is not very useful for speculative parallelization systems, as conflicts usually have a strict resolution determined by the need to preserve sequential semantics. Conflict avoidance, on the other hand, is a very worthwhile pursuit. One way to avoid conflicts and maintaining performance is creating a task scheduling component, responsible for scheduling speculative tasks in an efficient way. And this task scheduling component will make decisions based in part in past conflict history, collected by a data collector component. Interestingly enough, this conflict-aware task scheduler is also applicable to the case of software transactional memory, generalizing the concept of contention manager, that from now on can be seen as a very specific case of a conflict-aware scheduler that makes decisions based only on the set of currently conflicting tasks (as opposed to all running tasks, and tasks awaiting execution).

3.1 Extending Jaspex

To evaluate my thesis regarding conflict-aware scheduling in the scope of speculative parallelization systems, I extended Anjo’s Jaspex system [2, 1], a speculative parallelization system running on top of the JVSTM [5, 6].

To implement conflict-aware task scheduling, I first incorporated a data collector component. Jaspex, as of now, speculates on method executions only; therefore, the data collector creates a map of conflicting methods, where, upon each conflict, a pair of conflicting methods is registered. The JVSTM also had to be slightly changed, so that it would report for each conflict at least one pair of conflicting transactions (in the standard implementation of JVSTM this information was not made available externally, although the information is available internally). This data collector component was implemented in a lock-free way, so that it can run in its own thread and be highly scalable, presenting no contention point even for hundreds of threads. This was achieved by the use of lock-free (test-and-set) hash tables on one hand, and on the other hand by allowing a small margin of error. As the scheduler uses these data to make heuristic decisions, the fact that it may read data with a small margin of error results in at most a bad scheduling decision; this is definitely a small tradeoff to pay. The margin of error we have to account for results from the simple fact that the data may evolve while the scheduler reads it (as it does not lock the data), and thus may obtain a reading that does not correspond to a consistent view of the data at any given time point. Fortunately, the resulting deviations are small, as the scheduler reads the data at about the same rate as it is modified (which is directly related to task throughput).

3.2 Extending the STM Bench7

To also obtain validation on my thesis regarding conflict-aware scheduling in generic software transactional memory systems, I extended the STM Bench7 benchmark [9], a standard benchmark for evaluating software transactional mem-
ory runtimes. As this benchmark had already been adapted by other researchers to run on the JVSTM for evaluation purposes, this was my starting point.

To extend the benchmark, I had to modify the STMBench7, so that it would present a slightly different behavior. In the original benchmark, each thread continually generates and executes random operations, but there is never an excess of tasks available, as each thread only generates a task after finishing its current one. I modified this behavior, so that instead of one task being generated for each thread that goes idle, a batch of $N$ tasks is generated at the beginning of the benchmark, and then another batch is generated every time the current batch runs out. For my experiments I used the $N$ value of 2000, although I also briefly experimented with other numbers, such as 100, 500 and 1000, and came to the conclusion that this number had practically no influence in the results (as long as it stayed well above the number of available threads).

After this modification, I added a data collector component, collecting conflict data in the same way as in Jaspex, by typifying the operations available in the benchmark and registering a map of conflicts. I then plugged in the same scheduler component I had developed for Jaspex.

### 3.3 A Simple Scheduling Policy

For the conflict-aware schedulers, I developed a simple scheduling policy, that was used for all the experimental results reported in the next section. This policy is the No-Conflicters policy, and uses a simple and draconian rule: while a candidate task has an enemy in the thread pool – that is, a task with which it has conflicted in the past – it is not put into execution. In this way, tasks that have conflicted in the past are never again put into simultaneous execution.

### 4. EXPERIMENTAL RESULTS

The conflict-aware scheduler, using the No-Conflicters policy, was tested in two different scenarios: in a suite of programs automatically parallelized by Anjo’s Jaspex, and in the modified STMBench7 framework, running on the JVSTM.

#### 4.1 Results on Jaspex

##### 4.1.1 Experiments

The experiments with Jaspex were run on the Phobos machine, a machine with two quad-core Intel Xeon CPUs E5520 with hyperthreading. I ran the tests with eight threads, one per core. Each run consisted of executing a program from start to end in eight iterations, allowing three iterations for warmup and then taking the average execution time of the remaining five iterations. Each program was run in this way twice, once under the control (and thus parallelized by) the regular Jaspex system, and once under the control of the extended Jaspex system.

I started by running a small, artificial benchmark, developed to have a number of methods that conflict in pairs (method A conflicts with B, method C with D, etc). The parallelized version would put all of them into execution simultaneously, and therefore the influence of a smart scheduling policy should become evident over time. The results of using the extended Jaspex versus plain Jaspex on the artificial benchmark were not very encouraging, though. In the regular Jaspex the average time spent per iteration of the loop was of 27.2 seconds; in the extended Jaspex this same measurement amounted to 24.5 seconds, a speedup of about 11%. I also obtained results on several benchmarks, among them non-parallel benchmarks of the DaCapo suite of benchmarks [3, 4], the Java Grande benchmarks [16], and the JatMark benchmark [17]. I was unable to obtain speedup on any of these benchmarks, although I did not obtain any slowdown either.

#### 4.1.2 Discussion

Although the speedup obtained for the artificial benchmark is low, the results point out that there are at least some consistent gains to be obtained by adding conflict-aware task scheduling to Jaspex. In fact, plugging in an ideal schedule for this benchmark in the extended Jaspex gives us a speedup of about 73% relative to the run on regular Jaspex, so there is a big margin for improvement, namely for better scheduling policies.

#### 4.2 Results on STMBench7

##### 4.2.1 Experiments

The experiments on the modified STMBench7 benchmark were run in two different machines: Phobos, a machine with two quad-core Intel Xeon CPUs E5520 with hyperthreading (the same one used for the experiments with Jaspex), and Azul, an Azul Systems’ Java Appliance with 208 cores. On each of these machines, I ran the benchmark with three differing workloads - a workload consisting majorly of read-only operations (read-dominated workload), another consisting majorly of write operations (write-dominated workload), and a third where both kinds of operations are roughly at an equilibrium (read-write workload). Each workload was run for two minutes, with different runs for increasing amounts of threads, in powers of two. The output of these runs is an average throughput, in operations per second.

Figure 1 and Table 1 show the results obtained when running the benchmark on Phobos, and Figure 2 and Table 2 show the results obtained on Azul.

##### 4.2.2 Discussion

For the Phobos machine, in the read-dominated workload, there are hardly any conflicts (in fact, in the JVSTM, read-only transactions never conflict). Therefore, it is natural that the read-dominated workload exhibits degraded performance with the added scheduling, given the extra computing work that ends up being useless in the absence of conflicts. We can also see that after the number of threads exceeds the number of physical processors available, there is a loss of performance in both scenarios, due to wasteful context switching forced on the operating system.

For the read-write workload, there is a similar shape, with the same loss in performance after the number of threads exceeds the number of processors. However, here the conflict-aware task scheduling achieves greater throughput, with a gain of 32% at its peak (at 4 threads). For any number of threads, the benchmark with added scheduling is able to prevent conflicts between conflict-prone tasks, and thereby
Table 1: STM Benchmark data for the three types of workload in Phobos, for the regular benchmark and with added scheduling (+S). Average throughput is in operations/second.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Read</th>
<th>Read+S</th>
<th>Read-Write</th>
<th>Read-Write+S</th>
<th>Write</th>
<th>Write+S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3676</td>
<td>4009</td>
<td>2710</td>
<td>2728</td>
<td>2351</td>
<td>2463</td>
</tr>
<tr>
<td>2</td>
<td>7402</td>
<td>7436</td>
<td>4983</td>
<td>5422</td>
<td>3180</td>
<td>4302</td>
</tr>
<tr>
<td>4</td>
<td>12822</td>
<td>13480</td>
<td>6918</td>
<td>9138</td>
<td>3736</td>
<td>4034</td>
</tr>
<tr>
<td>8</td>
<td>20827</td>
<td>18081</td>
<td>7518</td>
<td>7302</td>
<td>3269</td>
<td>3017</td>
</tr>
<tr>
<td>16</td>
<td>18478</td>
<td>12055</td>
<td>4575</td>
<td>5807</td>
<td>1874</td>
<td>2900</td>
</tr>
</tbody>
</table>

Table 2: STM Benchmark data for the three types of workload in Azul, for the regular benchmark and with added scheduling (+S). Average throughput is in operations/second.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Read</th>
<th>Read+S</th>
<th>Read-Write</th>
<th>Read-Write+S</th>
<th>Write</th>
<th>Write+S</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>217</td>
<td>222</td>
<td>182</td>
<td>187</td>
<td>148</td>
<td>146</td>
</tr>
<tr>
<td>2</td>
<td>466</td>
<td>443</td>
<td>298</td>
<td>364</td>
<td>177</td>
<td>213</td>
</tr>
<tr>
<td>4</td>
<td>870</td>
<td>740</td>
<td>414</td>
<td>492</td>
<td>190</td>
<td>227</td>
</tr>
<tr>
<td>8</td>
<td>1500</td>
<td>1033</td>
<td>475</td>
<td>474</td>
<td>185</td>
<td>227</td>
</tr>
<tr>
<td>16</td>
<td>1618</td>
<td>1725</td>
<td>447</td>
<td>511</td>
<td>167</td>
<td>274</td>
</tr>
<tr>
<td>32</td>
<td>1594</td>
<td>1324</td>
<td>407</td>
<td>528</td>
<td>167</td>
<td>267</td>
</tr>
<tr>
<td>64</td>
<td>1651</td>
<td>1761</td>
<td>380</td>
<td>578</td>
<td>164</td>
<td>270</td>
</tr>
<tr>
<td>128</td>
<td>1156</td>
<td>1049</td>
<td>302</td>
<td>601</td>
<td>132</td>
<td>257</td>
</tr>
<tr>
<td>256</td>
<td>625</td>
<td>1181</td>
<td>190</td>
<td>558</td>
<td>74</td>
<td>274</td>
</tr>
</tbody>
</table>

Figure 1: STM Benchmark runs for three types of workload in Phobos, with (red) and without scheduling (blue). In the xx-axis we have the number of worker threads, and in the yy-axis we have the average number of operations per second.
Figure 2: STMBench7 runs for three types of workload in Azul, with (red) and without scheduling (blue). In the xx-axis we have the number of worker threads, and in the yy-axis we have the average number of operations per second.

stays always at or above the level of throughput of the regular benchmark.

The write-dominated workload is even more interesting, with a peak gain of 35% at two threads (similar to the read-write workload). However, and due to the high number of conflicts for this workload, at a high number of threads few tasks are selected for simultaneous execution (fewer than the number of available threads, as the scheduling policy has the option of not putting anything into execution), and therefore the throughput decreases much less than in the regular benchmark.

The results obtained for Azul are a great improvement over those obtained in Phobos. In the read-dominated workload, we have a scenario akin to the scenario on the Phobos machine. In the read-write workload and in the write-dominated workload, however, there is a consistent advantage for the scheduling benchmark. Not only are the performance gains much more significant than the results on Phobos, with a peak performance gain of 195% for 256 threads on the read-write workload, and a peak performance gain of 288% for the write-dominated workload, but adding conflict-aware task scheduling succeeds into turning a trend of descending performance (as the number of threads increase) into a trend of ascending performance.

These results show that conflict-aware task scheduling can have a great influence on performance on generic transactional memory systems, even with a naive scheduling policy such as the No-Conflicters policy. The greater the probability of conflicts, the greater the performance gains obtained with their prevention.

5. RELATED WORK

5.1 Software Transactional Memory

Besides the JVSTM [5, 6], which I used for the simple reason that Anjo’s Jaspex is built on top of it, there are other software transactional memory frameworks available, written in different programming languages and with different properties. Among the first unbounded (and therefore suited for object oriented programming) software transactional memories proposed was the Dynamic Software Transactional Memory (DSTM). Written in C++, it was first presented by Herlihy et al. in [11]. It was developed with the goal of going beyond the limitations of hardware transactional memory, and is still being actively researched. Herlihy was also the inventor of the concept of contention management, and the DSTM has been the prime research tool for research on contention management (such as the research conducted by Scherer and Scott in [14]).

Another relevant software transactional memory is Deuce [12]. Also written in Java, it implements the transactional model by mediating the access to fields; it thus requires instrumenting the whole Java runtime. Being written in Java and being among the most performant software transactional memories available, along with the JVSTM, it is a prime candidate for experiments in speculative parallelization systems.

5.2 Speculative Parallelization

Speculative parallelization, first presented as thread-level speculation, was shown by Steffan and Mowry in [18] to be able to uncover parallelism that cannot be exploited by parallelizing compilers that rely on static analysis to rule out dependency violations. Since then, thread-level speculation has been applied with success to loops [19, 8, 10], methods [7], and even coarser-grained sections of code [13]. In general, each author has developed his own speculation mechanism, from hardware supported speculation to simple copy-and-update schemes. This has resulted in the proliferation of speculative parallelization support techniques in the literature, of which I provide some examples.

Tian et al. [19] use thread-level speculation applied to loops,
using a copy-and-update scheme for speculative execution; speculative threads, which are spawned by the main thread, obtain memory copies of the data that they need to execute, and upon successful speculation (speculation that does not incur in any conflicts), the written data gets copied to the main thread.

Chen and Olukotun [7] apply thread-level speculation to methods, and obtain their results running the modified applications on simulated speculative hardware.

Kulkarni et al. [13] use an approach that is essentially based on developing and using optimistic parallel library classes in irregular programs – programs that have a complex memory access behavior, and therefore are also difficult to parallelize using static data dependency analysis techniques.

All of these approaches have helped their authors parallelize a number of programs and obtain significant speedups. Unfortunately, the speculation schemes are not generic enough to be able to provide a consistent working model for programmers, researchers, or automatic parallelizers (this is at least partially shown by the lack of reuse of these schemes in the scientific literature). The fact that Jaspex, an all-purpose automatic parallelizer, is built on top of a software-transactional-memory–based speculation support shows at least the commitment to moving towards more generic schemes.

6. CONCLUSIONS AND FUTURE WORK

I have shown that conflict-aware scheduling is a valid technique for enhancing the performance of transactional memory systems, even when run with a basic scheduling policy. The same, in principle, should be true of speculative parallelization systems run on software transactional memory support, although the results I obtained for this field are far from conclusive. As state-of-the-art speculative parallelization systems improve and become able to speculate on many tasks simultaneously, it will be possible to better assess the value of conflict-aware scheduling in this domain.

A big design space has been opened up by this research, and I leave a number of topics for future lines of research. The first is the validation of this approach on other performant transactional memory frameworks, with Deuce [12] being a forefront candidate. Another topic is the collection and incorporation of information collected and estimated about the tasks, such as task duration, range of data accessed, abort-to-commit ratio, among others, to allow more informed scheduling decisions. Finally, better and more sophisticated scheduling policies can be developed; the one I used throughout my work is just a starting point, and it should not be hard to obtain more efficient schemes, leading to even better results.

7. REFERENCES


implementation, pages 211–222, New York, NY, USA, 2007. ACM.


