A Study of a Transactional Parallel Routing Algorithm

Ian Watson, Chris Kirkham and Mikel Luján
The University of Manchester
School of Computer Science

Abstract

Transactional memory proposes an alternative synchronization primitive to traditional locks. Its promise is to simplify the software development of multi-threaded applications while at the same time delivering the performance of parallel applications using (complex and error prone) fine grain locking.

This study reports our experience implementing a realistic application using transactional memory (TM). The application is Lee's routing algorithm and was selected for its abundance of parallelism but difficulty of expressing it with locks. Each route between a source and a destination point in a grid can be considered a unit of parallelism. Starting from this simple approach, we evaluate the exploitable parallelism of a transactional parallel implementation and explore how it can be adapted to deliver better performance. The adaptations do not introduce locks nor alter the essence of the implemented algorithm, but deliver up to 20 times more parallelism. The adaptations are derived from understanding the application itself and TM. The evaluation simulates an abstracted TM system and, thus, the results are independent of specific software or hardware TM implemented, and describe properties of the application.

1. Introduction

With the promise of large scale multi-core processors providing significant amounts of thread level parallelism, there has been much interest in how they might be programmed. Multi-threaded programs usually co-ordinate their use of shared memory using traditional synchronization primitives such as locks and/or barriers. These low level synchronization primitives can ensure serialized access and execution for critical sections of programs. However, serialization of parallel programs is clearly undesirable except where it is absolutely necessary. In complex programs it is difficult to ensure that the use of locks or barriers does not lead to over serialization. Worse, it can be difficult to ensure the correctness of such programs, and problems such as deadlock can result.

Transactional memory (TM) is a programming model which promises to overcome these inefficiencies and difficulties. A program is expressed as a collection of parallel transactions which either complete and commit their updates to global memory or fail because they observe that other transactions have committed changes to memory which invalidate the data values on which they have been operating. There have been proposals for support of this model either in software only or by providing hardware which assists the transactional processes [15].

A major claim for TM programming is that it allows parallel programs to be written easily to solve problems which would otherwise be complex to implement using traditional synchronization primitives. However, there have been few studies of practical TM programming which examine and verify this claim. Many TM systems have been evaluated by modifying programs which have been written using synchronization. For example, Java synchronized methods can often be turned directly into transactions without any need to otherwise modify the code. One limitation of this approach is that it uses programs where the parallel structuring has already been done and therefore does not properly examine the practicality of TM programming. A further limitation is that the synchronized sections in such programs are kept deliberately short whereas the TM style may naturally lead to long running transactions [4].

We therefore decided to look for a practical application which should clearly be able to exploit significant parallelism, but which would be complex to implement using traditional synchronization. The intention was to gain insight into the real issues in writing TM programs as well as evaluating whether the resulting program would exhibit worthwhile parallelism and performance.

The application that we chose was circuit routing using Lee's algorithm [16]. This is a well known technique which is applicable to either integrated circuit or printed circuit board routing. In reality, industrial strength routers use more elaborate techniques than Lee's algorithm. Nevertheless, we have developed a TM routing program which makes a realistic job of producing a practical solution. This

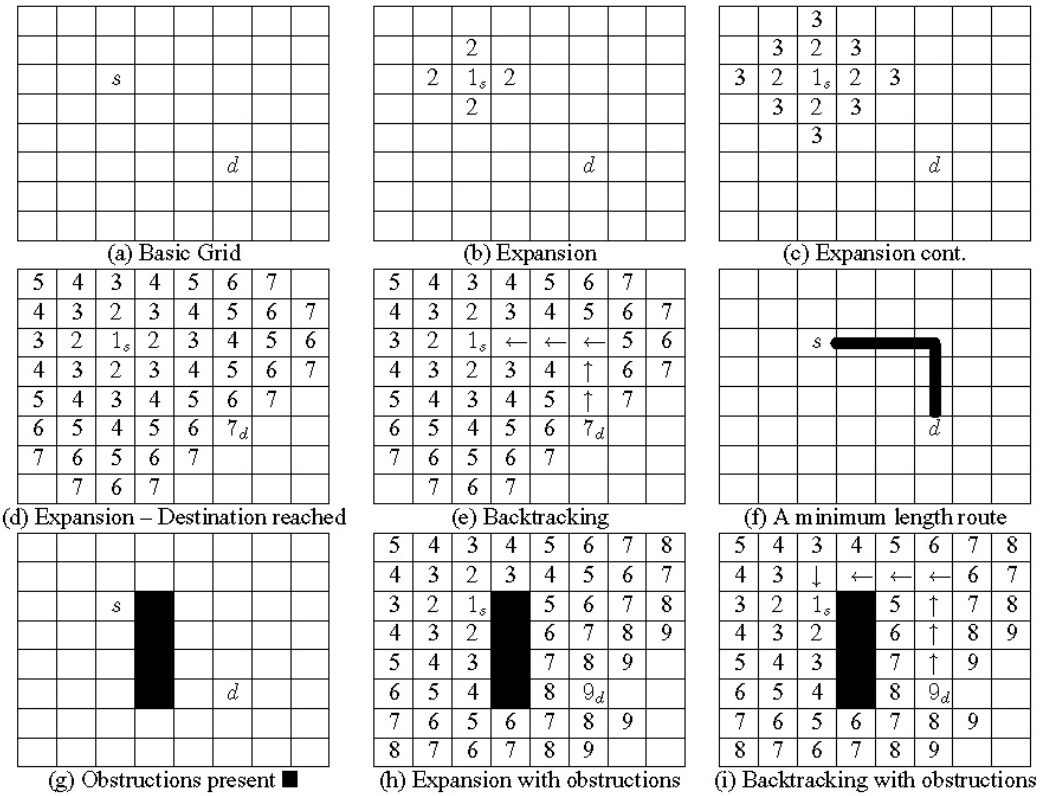

Figure 1. Phases in Lee's algorithm.

TM program is evaluated by taking a real circuit board layout. This paper presents a description of the algorithm (Section 2), and an analysis of the difficulties of parallelizing it using traditional synchronization (Section 3). Section 4 presents a description of a TM solution based on defining a transaction as the process of finding a complete route. This first attempt can be considered as the simplest parallelization strategy and its performance evaluation (see Section 6) shows an average exploitable parallelism of about 5 from a theoretical maximum of 1506. All the evaluations yield results that are TM implementation independent. Guided by the high levels of transaction aborts, the TM program is adapted by privatizing one data structure (see Section 4), but this still yields a poor average exploitable parallelism of about 7. In both TM programs, the software development is very simple, but the performance is unsatisfactory. To produce a TM program that improves the exploitable parallelism up to about 100, the defined transactions need to reduce their exposure to abortions. The key insight is understanding that the solution to the routing problem is a set of routes which do not contain any intersection points. Therefore, the minimum exposure for a transaction that generates a complete path between a source point and a destination point is simply those points contained in the complete path; no more, no less. A discussion of the proposed and implemented TM programs and related work is presented in Section 7.

2. Background - Lee's Algorithm

Circuit routing is the process of producing an automated interconnection of electronic components. The components may be transistors on an integrated circuit, logic elements on an FPGA or packages containing integrated circuits placed on a printed circuit board, for example a PC motherboard.

In its simplest form, the problem can be reduced to that of joining points on a two dimensional grid which represents the circuit layout. Lee's algorithm guarantees to find a shortest interconnection between two points using the technique illustrated in Figure 1. Starting at the source point, the grid points are numbered by expanding a wavefront until the destination is reached (see Figure 1(a)–(d)). At each stage of the expansion any grid point in the wavefront marks its unnumbered neighbors with an increment of its value. Once the destination point has been reached, a route is traced back to the source by following any decreasing sequence of num-

```
Grid global;

for i in number of routes {
    Expand from source to destination;
       // reads and writes to global
    Backtrack from destination to source;
       // reads and writes to global
    Reset Expansion;
}
```

Figure 2. Pseudo code for Lee's algorithm.

bered grid points (see Figure 1(e)). This backtracking process can follow any one of a number of routes and a practical implementation will impose a particular strategy. For example, it may be desirable to follow straight lines where possible rather than continually change direction.

A practical circuit will contain many grid interconnection points which cannot be used for routing unless they are part of the circuit. In addition, once a route has been determined, the grid points that it uses are occupied and cannot be used by others. Expansions must therefore spread only into free points and flow around occupied ones. Figure 1(g)–(i) shows an example of this and how the backtracking process can nevertheless find a minimum length connection around these obstructions. Figure 2 illustrates Lee's algorithm using pseudo code.

There are many refinements that can be made to the basic algorithm. For example, most practical circuits have multiple routing layers and the algorithm must be extended to three dimensions to model this. A route may need to cross to another layer using a *via* to avoid hitting a dead end on its current layer. It is also possible to mark grid points with initial weights which are included in the numbering process. By this means, it is possible to ensure that routes are encouraged to prefer certain areas. This can, for example, avoid routes crowding interconnection points unless they are connected to them. The details of these refinements are unimportant to comprehend the routing problem and the TM solutions described. Nevertheless, to tackle real practical routing problems, our implemented programs do use these techniques.

3. Parallelizing Lee's Algorithm

A real circuit will contain orders of magnitude more connections than the number of cores which will be available in multi-core processors for sometime to come. It is clear that there ought to be significant opportunities for exploiting parallelism by finding multiple routes concurrently. Given interconnections at widely separated locations in a grid, multiple wavefront expansions and subsequent backtracking could occur independently in parallel. The natural implementation, and that usually used by serial programs, would use a single array to represent the circuit grid points. Both the expansion and the backtracking can use this single array, although this requires that any unused grid points are reset after the backtracking.

A parallel version could also use a single grid array as long as the expansions do not overlap. Assuming traditional synchronization primitives, what are the options? We could have a global lock for the grid so that only one expansion can occur at once, but this would destroy the parallelism. We could use an array of locks to serialize access to individual grid points, but this degree of fine grain locking is likely to need complex protocols to avoid deadlock.

A better solution would be to use a separate grid for each parallel expansion. We would still need to access a global shared version of the grid to determine grid point occupation, but now the expansions can proceed independently. However the backtracking phase is still an issue. We must trace a route back from destination to source and, as we do so, must record the usage of individual grid points in the global shared grid. We cannot allow two routes to occupy the same point and therefore any backtracking decision must be unique and global. Again, with traditional synchronization, a global lock for the grid could protect each backtracking point update. Or, we could again use an array of locks to serialize access to individual points but, as before, the complexity implications are serious.

Despite fine grain locking and separate expansion grids, we may not achieve a correct solution. If two expansions have overlapped and one backtrack has occurred then part of the other expansion is now invalid possibly leaving no valid backtrack. The only solution is to abandon the second attempt and restart the expansion.

It may be possible to program a correct solution with traditional synchronization, but it should be apparent that this will not be easy. Suffice it to say that we have failed to locate published versions of Lee's algorithm which exploit parallelism at the route level. However there are some which partition the grid among processors to parallelize Lee's algorithm [28].

4. Transactional Memory & Lee's Algorithm

It is apparent that there is something naturally transactional about this problem. We can treat each route as an independent transaction. Each routing transaction can perform its own expansion, backtrack, and then try to commit the route it has found. If any of the grid points used by the route have subsequently been utilized and committed by another route, then the transaction must be abandoned and restarted. However, it is important to realize that, in this case, the detection of interference, abandonment and restarting are fundamental functionality provided by TM.

```
Grid global;

forall routes {
   atomic {

      Expand from source to destination;
         // reads and writes to global
      Backtrack from destination to source;
         // reads and writes to global
      Reset Expansion;
   }
}
```

Figure 3. The simplest transactional Lee's algorithm — Lee-TM.

```
Grid global;

forall routes {
   atomic {
      Grid local;
      Expand from source to destination;
         // reads from global, writes to local
      Backtrack from destination to source;
         // reads from local, writes to global
      // no longer needed Reset Expansion;
   }
}
```

Figure 4. Transactional Lee's algorithm with privatization — Lee-TM-p.

There is no need to program it explicitly as would be required in a solution which used locking.

The important question is how easy is it to produce the TM program? A serial version of the problem can be expressed very easily. We will assume, for the moment, that this is the simplest of serial programs which uses a single grid array for both expansion and backtracking (see Figure 2). Although it is not apparent from this simple formulation of the algorithm, it is usual to order the routes in ascending order of length. This ensures that longer routes, which naturally have more alternatives, do not displace shorter ones from their natural positions.

In a nutshell the semantics of TM are as follows. Each transaction will have associated a set of reads and writes that have been executed, as well as a means for rewinding the program state. When a transaction commits it will atomically validate its read set, and then ensure visibility of the committed write set. Any transaction which fails the validation must abort and will restart. Any new read which occurs will obtain the value written by the latest commit. The natural expression of the problem did not lead us to *nested transactions* [15], and thus the paper omits this issue at this point. We have not assumed any detailed programming notation other than the usual `atomic` programming construct [15]. In the following examples we use simple pseudo code to express the relevant algorithms. The simplest TM conversion of the serial program appears in Figure 3.

All we have done is to assume that all routes are started in parallel, and enclosed the loop body in an atomic statement. In practice, the `forall` construct would need to involve a work queue from which individual routes were taken and this itself is a shared structure which would need to be handled, possibly transactionally. However, this is of minor importance to the main algorithm and we have therefore neglected this detail. Consider what will happen when the program executes.

As a route expands, it must read each neighbor point on the global grid to check whether it is empty. It will then write its expansion values to the grid points until the destination is reached. The backtracking will then read some of the expansion values and write the points which the route will occupy. When the backtracking has finished, the transaction will commit, assuming that it has passed the validation. At this point, the route points it has committed and all the grid points it has written to in the expansion will certainly be visible by other threads.

Here we see the first problem of a simple approach to TM parallelization. Because a single global grid was used in the serial version for both expansion and backtracking, the expansion phase updates global shared variables. These updates will be significantly more extensive than the selected route and will cause many unnecessary aborts. In practice, the expansion phase generates temporary information only relevant to each route operation and making its updates externally visible is not only unnecessary, but seriously harmful.

If we modify the program to use a local grid array during the expansion (for example as a local variable) then writes to it from each transaction will not overlap and interfere with others. In fact it is clear that any TM model should incorporate local variables whose values do not need to, and should not, be updated on commit. This modification is equally applicable to a serial version of the program and removes the need for the grid reset operation, although the temporary local grid may need initializing. Figure 4 illustrates the TM program with the privatization changes.

5. Further Optimization

As we will see in the evaluation, Section 6, the use of privatization produces a modest increase in performance. However, it is worth examining whether we have produced an optimal TM program. To minimize the possibility of a transaction aborting, we should try to ensure that only those reads and writes which affect the outcome of a transaction's

computation can cause an abort.

During the expansion phase of the algorithm a transaction will read from the global grid to check whether it can expand into a particular grid point or whether it is blocked either by a connection point or another route that has already been committed. However, the function of the expansion is to mark all possible minimum length routes. If the backtracking phase chooses a route which does not use a particular grid point then it is irrelevant whether or not that point was marked (read and written) incorrectly as usable during the expansion. An individual routing operation takes a snapshot of the global grid and starts its routing operation. When it has found a route and wishes to commit, the only thing that should prevent this is another route having committed and used a common grid point. When considering whether to abort a routing transaction, the only relevant issue is whether another committed transaction's route contains grid points which are also part of the new route!

More formally, when detecting TM interference and the need to abort, we can discard all grid points which are not in the intersection of the read set, the write set, and those grid points that constitute the route for a committing transaction. In practice, in this algorithm, anything in the write set of the local transaction is also in its read set, so we only need to consider the intersection of the write sets. This clearly is not covered by the simple semantics of the TM model described above. We will return to this in a later discussion (see Section 7). The TM program which uses this insight with respect to the write set is referred to as Lee-TM-p-ws.

6. Evaluation

6.1. Description of the Experiments

The behavior of a routing algorithm can be influenced significantly by the circuit layout. A set of connections which are too regular will not result in the degree of contention encountered on a real layout. To perform the study on a realistic scenario, we have used a two layer printed circuit board which contains a microcoded microprocessor. This layout was used in a project to develop advanced routing software using a variety of algorithms [26]. Although it is a fairly old design it contains a significant amount of random wiring which makes the layout non-trivial.

We started by developing simple two-layer routing software written in Java together with a program to extract the layout and connection information from the board description and produce input for our router. The layout contains over 3000 connections points and 1506 interconnections.

We wanted to keep the routing program simple, as the study is concerned with understanding the behavior of a TM program rather than developing sophisticated routing software. However, it was necessary, in order to achieve successful and realistic routing of the example circuit, to add a certain amount of refinement in both the expansion and backtracking phases of the algorithm. These are concerned with constraining the routes in certain ways so that the layout does not become a complete 'spaghetti' and their detail is unimportant. They do not alter the fundamental properties or the analysis that has been presented of potential problems and improvements.

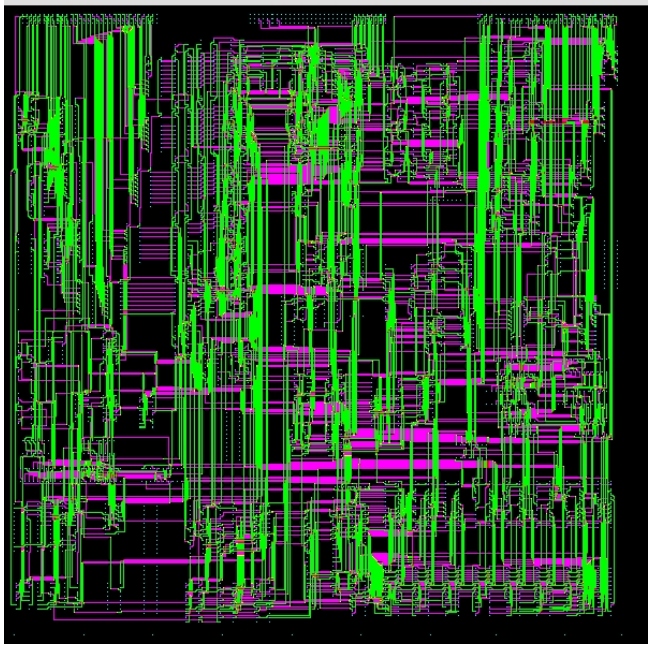

Figure 5. The test circuit layout used in the experiments.

The program reads in the connection point and routing data, and then sorts the routes by increasing length. It then routes them one by one until all have been completed, and finally produces a display of the test circuit. This is presented in Figure 5 to give an indication of the nature and complexity of the problem being solved.

It succeeds in routing all but 2 of the 1506 connections and, in the process, inserts approximately 3400 vias to change routing layers. To achieve good routing performance, it is generally advisable, on a 2 layer circuit, to restrict connections to run largely in one direction on one layer and orthogonally on the other. This might be expected to use approximately 2 vias per route although algorithm tuning might succeed in reducing this. We are not claiming that this is industrial strength routing software, but we believe that it is sufficiently realistic to represent a study of a real application. The programs and data are available for download from the website [1].

We considered using either a simulator for a hardware

based TM system or a software TM system running on a multiprocessor. However, either approach would have produced results which were specific to the characteristics of the particular implementation. Instead we wanted to estimate the potential for TM execution in an implementation independent manner. The approach we adopted was to take the serial versions of the programs and instrument them in a way which enabled the observation of interference without making assumptions about how that would be done in a real implementation. The instrumented version of the program runs serially and attempts the commits in the program order, but the overall effect is as though we were running the program transactionally with an infinite number of processing resources. The evaluation simulates an abstracted TM implementation and, thus, the results are independent of specific software or hardware TM implemented, and describe properties of the application.

All the transactions were executed, and each commit either succeeded or failed. Those that failed were then run again, and so on until all commits had succeeded. Thus the execution simulated a number of iterations in which each transaction got an opportunity, and at least one commit succeeded per iteration.

Within a transaction, where the code wrote to a shared data structure, the location and value to be written were recorded using a `Hashtable` and the writing deferred. Similarly locations where reads occurred from any shared data structure were remembered in a `Vector`. At the end of the transaction, the commit operation would check these against modifications made by already committed transactions in the same iteration. In the original transactional model, reads from such a location would cause the commit to fail. In the modified model, a recorded write to a location already written caused the commit to fail. Otherwise the commit succeeded, and the modifications were performed - and remembered in a `HashSet` for future commits in this iteration. All the transactions for which commits failed have to restart at the beginning, but they then reread the updated shared data structures. The instrumented version of the program runs serially, but the overall effect is as though we were running the program transactionally with an infinite number of processing resources.

The effect of having only a limited number of processors was simulated by taking a limited number of transactions off the queue for each iteration. When some transactions fail to commit, these jump to the front of the queue for the next iteration. This is needed to maintain the quality of the routing, which depends on the order.

For simplicity the simulation of TM execution batches transactions, so that each batch starts at the same time. In reality, transactions would start as processing resources become available. We also make no attempt to notice ahead of the commit that the transaction will need to abort. These experiments may therefore give a pessimistic view of the parallelism available.

6.2. Results

Three sets of results are presented: Lee-TM, Lee-TM-p, and Lee-TM-p-ws. The first uses only a single global grid for expansion and backtracking of all routes. Lee-TM represents the naïve TM program that we might produce if we started from a serial program and simply enclosed the expansion and backtracking phases in a transaction.

For Lee-TM, Lee-TM-p, and Lee-TM-p-ws, Table 1 shows, in the first row, the number of batched iterations needed to execute the program. In the second row, the table presents the number of routes which succeed in committing at the first batched iteration. In the third row the total number of commits attempted is reported as an indication of the amount of work.

For now consider only the second column labelled as Lee-TM (see pseudo code in Figure 3). The number of initially successful routes is low; 70 out of 1506. Furthermore, we only manage to exploit an average parallelism of about 5 (1506 routes / 305 iterations), but at the expense of nearly 60 times (89534 attempted commits / 1506 routes) the amount of work. This is because most of the writes in the expansion phases are interfering in the one global structure. This is not a very encouraging performance.

Batched iterations	305	227	14
Successful commits in 1st iteration	70	118	697
Total commits attempted	89534	53838	3774
	Lee-TM	Lee-TM-p	Lee-TM-p-ws

Table 1. Summary results for the Transactional Lee's Algorithm programs.

The second experiment replaces the use of a global grid with a local one for each expansion within a transaction as described in Figure 4. This TM program uses privatization and is labelled as Lee-TM-p in the third column of Table 1. The results show that the privatization has made some improvement. The average parallelism has risen to nearly 7, although the total work done is still 36 times the serial version.

Finally, we consider the TM program which uses our insight into the algorithm – the write set, which contains only those grid points selected during backtracking, is enough to capture all the information needed to decide whether to abort the transaction. This version was introduced in Sec-

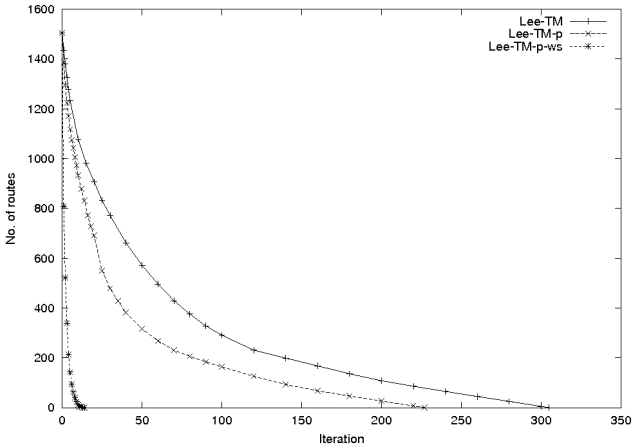

Figure 6. Graph of remaining routes to be completed as execution progresses.

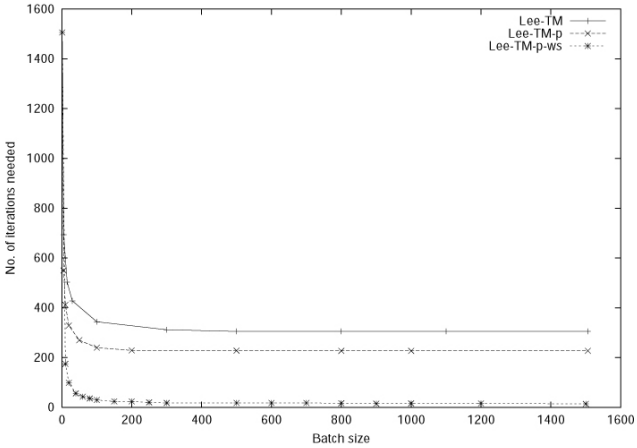

Figure 7. Graph of performance with different number of processors.

tion 5 and is labelled as Lee-TM-p-ws in the fourth column of Table 1. For Lee-TM-p-ws, the results in the table represent a dramatic increase in performance. Almost half the routes succeed in the first batched iteration and they all succeed within 14 iterations. The average parallelism jumps to over 100, and the number of commits attempted is only increased by a factor of 2.5 over the sequential program.

Figure 6 is a graph showing, for each of these three TM programs, how the numbers of routes remaining to be completed decreases as more batched iterations are executed. The results considering fewer processors at a time are also presented graphically in Figure 7. It shows how the number of iterations needed decreases non-linearly with the number of processors (batch size) available. As expected with 1 processor, 1506 iterations are needed. However, the descent in the number of iterations needed as the number of processors increases is initially quite rapid. Figure 8 shows how the amount of work done increases as more processors are used. This is quite gradual, especially for Lee-TM-p-ws. Together these results are very encouraging and suggest that it is possible to exploit significant amounts of parallelism in Lee's algorithm without excessive overhead.

7. Discussion & Related Work

A major claim for TM programming is that it makes relatively easy to parallelize programs because there is no need to consider the operational detail of the interaction among transactions. This enables TM programs to be composed with other TM programs, while traditional synchronization does not compose (see examples in [15, 18]). With TM, it is not a correctness issue if conflicts occur. The TM system simply aborts all but one of the transactions involved in the conflict and restarts them to try again. In certain scenarios TM programs can end up completely serialized, but they will still produce correct results. TM program examples have been constructed which produce infinite loops due to retrying transactions [3]. Larus and Rajwar [15] provide a comprehensive introduction to and survey of TM.

This study has taken an algorithm which is complex to parallelize using traditional synchronization primitives, and has shown that a highly parallel program can be achieved using transactions. However, the process required a little more thought than originally anticipated and a means to control the contents of the read and write sets for any transaction.

The discussion and related work are organized around the following questions:

I What other algorithms or benchmarks have been used in TM evaluations?

II What software design approach was followed to parallelize and improve TM execution if any? How does it compare?

III What TM constructs or mechanisms have been proposed to control the contents of the read or the write set for a given transaction?

IV What is the effect of aborts with respect to selected routes?

Question I — In 1977 Lomet [17] first described the concept of `atomic` and illustrated it with a shared Abstract Data Type (ADT) – a concurrent producer-consumer buffer. This paper did not include an implementation for

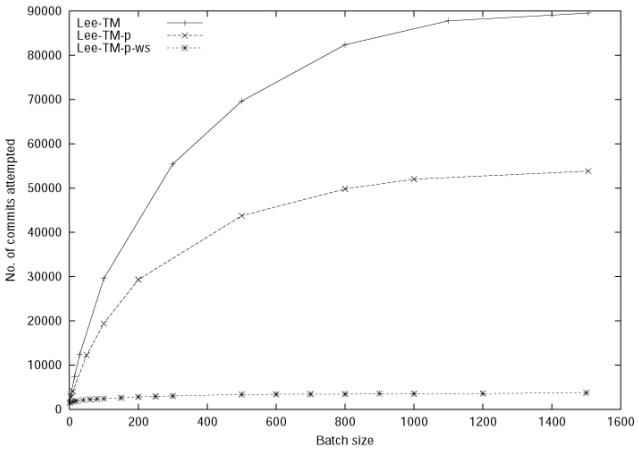

Figure 8. Graph of the number of transactions executed with different number of processors.

`atomic` and, thus, no performance evaluation was presented. Herlihy and Moss [13], and Shavit and Touitou [23] evaluated their TM implementations with benchmarks based also on ADT by measuring the throughput as the number of operations completed (different combinations of reads and writes) within a fixed time period. Both of these papers rely on simulated hardware. Herlihy *et al*. [12], and Harris and Fraser [11] present the first evaluations of TM, in this case software TM, running on real multiprocessors. Most of the TM papers surveyed by Larus and Rajwar [15], and also newer ones not covered in the survey [22, 9, 4] use these ADT *micro-benchmarks*. A different approach is to translate automatically already parallel benchmarks written in Java and C into TM benchmarks; e.g. a modified SPECjbb2000 [5, 4, 19, 7], some benchmarks from SPLASH-2 [27, 8, 6, 21, 19, 7], others from JavaGrande [19, 7], Linux 2.4.19 kernel and SPECjvm98 [2], and Berkeley DB [21, 8].

Chung *et al*. [7] present the most comprehensive study looking at 35 different TM benchmarks covering from mainly scientific computing (JavaGrande, SPLASH-2, NAS, and SPEComp), to commercial workloads (DaCAPO, and SPECjbb). These TM benchmarks were generated following the second evaluation approach; direct translation from parallel benchmarks. The performance evaluation provided a wealth of data with respect to size of transactions, read and write set sizes, nested transaction depth, and so on. Chung *et al*. [7] acknowledge a limitation of their methodology which is not able to generate the frequency of transactional aborts. Obviously our study does not have the same breadth, but our results are TM implementation independent, and the number of aborts is easily obtainable which has proved indispensable to analyze the different TM implementations of Lee's algorithm. To attempt to improve the performance of a TM program without the abort rate information eliminates one of the best indicators to identify excessive serialization.

One common limitation for all the TM evaluations referred to so far is that they use micro-benchmarks or benchmarks where the parallel structuring has already been done, and therefore do not properly examine the practicality of TM programming.

A theoretical study of using TM for the generation of meshes conforming with the Delaunay property is our closest relative [14]. Kulkarni *et al*. [14] describe the mesh generation algorithm and argue its suitability for TM. The mesh generation is not completed until every element in the mesh satisfies the Delaunay property. To those elements not conforming with the property, a refinement process is applied. An important feature is that the refinement only affects the immediate neighbors. Thus, refinements of elements far apart in the mesh can normally proceed in parallel. Kulkarni *et al*. [14] discuss a possible issue with TM conservative conflict detection mechanisms. The problem can be characterized as some conflicts detected at the memory level in reality may not be present at the application semantic level. The different TM implementations of Lee's algorithm can be interpreted as our steps to eliminate the conflicts at the memory level until only remain those conflicts defined at Lee's Algorithm level. This TM study of mesh generation does not provide TM implementations or performance evaluation. Spear *et al*. [25], inspired by [14] but independently, have developed a TM implementation of the Delaunay mesh generation algorithm relying on their software TM library. A very succinct mention reports that privatization is very attractive since 98% of the generation time is spent on data amenable to this optimization. Despite this favorable feature and due to the overhead of their software TM library, the mesh generation TM application suffers normally a 2 times slowdown. To sum up, the evaluation by Spear *et al*. [25] is specific to their TM implementation, the application differ significantly with respect to levels of contention (or aborts) and profitability from privatization, and at the moment there is not enough information to determine whether TM programming eased the parallelization.

In the last six months, the STMBench7 [10] and STAMP [20] benchmark suites have been proposed to evaluate TM implementations. STMBench7 is derived from a single threaded database benchmark, while the latest STAMP (release 0.9.4) includes three benchmarks parallelized from scratch using coarse grain transactions. Both publications [10, 20] evaluate specific TM implementations but do not report insights into the parallelization process using TM programming.

Question II — The naïve TM implementation of Lee's

algorithm was straightforward, and involved only the identification, as a transaction, of the code which shared the grid data structure. The changes, compared with the sequential program, were minimal. However, the performance results were not encouraging. A little thought revealed that some uses of the shared grid were unnecessary, and a reduction in conflicts (and a performance improvement) resulted from privatization. It should be emphasized that the program modifications were equally applicable to a serial version and would have little or no effect on serial performance. Had we started from that program the transformation into a TM version would have been equally simple. Unfortunately, although privatization delivered a slight improvement, the performance was hardly spectacular, demonstrating only a low degree of parallelism and much wasted work.

The next step delivered a more significant performance increase, but was less straightforward. The insight that only conflicts among grid points which are part of tentative or established routes are relevant to decide whether to abort a transaction requires a clear understanding of the particular algorithm. However, the resulting performance increase was spectacular and warrants a detailed analysis of the issues.

Herlihy *et al*. [12] described how reducing the amount of conflict among transactions can improve performance. They illustrated that conflicts at the memory level, for the implementation of an integer set using an ordered linked list, may not conflict at the application domain level. Accordingly, they propose *early release* which is a mechanism to remove elements from the read set associated with a transaction. Their evaluation uses a micro-benchmark based on the integer set and demonstrates a worthwhile performance improvement. Skare and Kozyrakis [24] extended the evaluation of early release to other micro-benchmarks. But they found that unless the ADTs were linear the performance improvements using early release were negligible. One possible view of the described Lee-TM-p-ws implementation is that it is an extreme and extended application of early release.

Carlstrom *et al*. [4], and Ni *et al*. [22] evaluate open nested transactions as a means to reduce conflicts, specially for long running transactions. Carlstrom *et al*. [4] examine the classes in the Java collections framework and use the concept of *semantic concurrency control* to eliminate irrelevant conflicts generated at the memory level. The implementation of semantic concurrency control requires open nested transactions. Ni *et al*. [22] describe a software TM implementation with support for open nested transactions, and the implementation is evaluated with two classes which are part of Java collections framework.

Question III — We have not concerned ourselves with how Lee-TM-p-ws might be expressed with TM constructs. Early release introduces a method `release()` that takes as a parameter the element to be removed from the read set associated with a transaction. Lee-TM-p-ws can be implemented by leaving intact the `atomic` construct around the main loop body, but releasing every read from the global grid during the expansion phase. During the backtracking phase, before writing to the global we need either (1) a new method `addToReadSet()` which takes as a parameter the grid point we are about to write to, or (2) we need to read points again from the global grid point so that these become part of the read set.

Open nested transactions could also be used to implement Lee-TM-p-ws, although we do not favor it. Again we leave intact the `atomic` construct around the main loop body. Every single instruction in the expansion phase becomes an open nested transaction. In addition a new open nested transaction is placed around the backtracking phase, and the write operations are redirected to the local grid. Then an extra closed nested transaction is needed. This transaction simply reads from the global grid those points that are tentatively part of the route. It will `abort()` if the read grid point have been mark in the global grid as being part of another committed route. Otherwise after having read those grid points, it will write the route in the global grid, and try to commit.

Alternatively, we would like to use only one `atomic` block around the extra closed nested transaction described above and eliminate the existing one around the main loop body. However, on abort the execution should restart from the beginning of the main loop body. The retry action encompasses more than the scope of the transaction.

The proposed usage of early release, open nested transactions, and Lee-TM-p-ws make a conscious decision to ignore the effect of certain memory operations. If we are wrong, it will probably result in an incorrect program. This clearly compromises a major advantage of the transactional approach: a program can be easily parallelized using transactions without affecting correctness. But, given the performance improvements, we may well be prepared to accept this added complexity.

Question IV — We have observed a dramatic performance increase at the expense of a more complex transactional program. There are some details of Lee's algorithm and its implementation which ought to be mentioned for completeness. It was stated previously that, in a serial implementation of Lee's algorithm, routes are normally sorted in order of increasing length. This ensures that simple short routes are completed early and longer routes, which have more options to find optimal connections, do not interfere with them. In our simulation, this ordering is initially maintained as the routes are processed in their original sorted order. However, any routes which fail due to conflict may now get 'overtaken' by longer ones and this possibility is clearly present in any truly parallel version. The major con-

sequence of this is that the number of layer changes (vias) needed for a successful overall routing increases. We will not present the detail of this but observe that this appears as a factor of two or three in our optimized results depending on exact configuration. In a real implementation, it might be necessary to address this problem by more careful scheduling.

We have not attempted to generalize the finding to other algorithms, but arguably it is likely that there are others with similar features. In situations where there is a shared space examined in parallel with a view to occupying or performing modifications on that space, then part of or all of the examination may not be relevant to the final outcome. This may occur, for example, in the areas of image processing and planning.

8. Summary

A major claim for TM programming is that it allows parallel programs to be written easily to solve problems which would otherwise be complex to implement using traditional synchronization primitives. However, there have been few studies of practical TM programming which examine and verify this claim.

This study reports our experience implementing a realistic application using transactional memory. Lee's algorithm is a well known technique which is applicable to either integrated circuit or printed circuit board routing. Without the help of an existing parallel implementation, we have developed parallel TM routing programs which make a realistic job of producing practical solutions. These TM programs have been evaluated by taking a real circuit board layout. This first TM attempt considers finding each route as a transaction, and its performance evaluation shows an average exploitable parallelism of about 5 from a theoretical maximum of 1506. All the reported results are TM implementation independent. Guided by the high levels of transaction aborts, the TM program is adapted by privatizing one data structure, although this still yields a poor average exploitable parallelism of about 7. In both TM programs, the software development is very simple, but the performance is unsatisfactory. To produce a TM program that improves the exploitable parallelism up to about 100, the defined transactions need to reduce their exposure to abortions. The key insight is understanding that for Lee's algorithm the minimum exposure for a transaction that generates a complete path between a source point and a destination point is simply those points contained in the complete path.

9. Acknowledgements

This work has been supported by the EPSRC grant EP/E036368/1.

References

[1] http://www.cs.manchester.ac.uk/apt/projects/TM/LeeRouting.

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. *IEEE Micro*, 26(1):59–69, 2006.

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

[4] B. D. Carlstrom, A. McDonald, M. Carbin, C. Kozyrakis, and K. Olukotun. Transactional collection classes. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 56–67, 2007.

[5] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–13, 2006.

[6] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded page-based transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 347–358, 2006.

[7] J. Chung, H. Chafi, C. C. Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. The common case transactional behavior of multithreaded programs. In *Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture*, pages 266–277, 2006.

[8] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international Conference on Architectural Support for Programming Languages and Operating Systems*, pages 336–346, 2006.

[9] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 21–33, 2007.

[10] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: a benchmark for software transactional memory. In *Proceedings of the 2007 Conference on EuroSys*, pages 315–324, 2007.

[11] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the 18th annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, pages 388–402, 2003.

[12] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual Symposium on Principles of Distributed Computing*, pages 92–101, 2003.

[13] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[14] M. Kurkarni, L. P. Chew, and K. Pingali. Using transactions in Delaunay mesh generation. In *Proceedings of the Workshop on Transactional Memory Workloads*, pages 23–31, 2006.

[15] J. Larus and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2007.

[16] C. Y. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, EC-10:346–365, 1961.

[17] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of an ACM Conference on Language Design for Reliable Software*, pages 128–137, 1977.

[18] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research directions in concurrent object-oriented programming*, pages 107–150. MIT Press, 1993.

[19] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 53–65, 2006.

[20] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th annual International Symposium on Computer Architecture*, pages 69–80, 2007.

[21] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logTM. In *Proceedings of the 12th international Conference on Architectural Support for Programming Languages and Operating Systems*, pages 359–370, 2006.

[22] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 68–78, 2007.

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

[24] T. Skare and C. Kozyrakis. Early release: Friend or foe? In *Proceedings of the Workshop on Transactional Memory Workloads*, pages 45–52, 2006.

[25] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. Technical Report TR 915, Department of Computer Science, University of Rochester, February 2007.

[26] T. D. Spiers and D. A. Edwards. A high performance routing engine. In *Proceedings of the 24th ACM/IEEE conference on Design Automation*, pages 793–799, 1987.

[27] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 34–48, 2007.

[28] I.-L. Yen, R. M. Dubash, and F. B. Bastani. Strategies for mapping Lee's maze routing algorithm onto parallel architectures. In *Proceedings of the 7th International Parallel Processing Symposium*, pages 672–679, 1993.