Revisiting Deterministic Multithreading Strategies*

Jörg Domaschka, Andreas I. Schmied, Hans P. Reiser, Franz J. Hauck

Ulm University
Institute of Distributed Systems
James-Franck-Ring O-27, 89069 Ulm, Germany
{joerg.domaschka, andreas.schmied, hans.reiser, franz.hauck}@uni-ulm.de

Abstract

Deterministic behaviour is a prerequisite for most approaches to object replication. In order to avoid the nondeterminism of multithreading, many object replication systems are limited to using sequential method execution. In this paper, we survey existing application-level scheduling algorithms that enable deterministic concurrent execution of object methods. Multithreading leads to a more efficient execution on multiple CPUs and multi-core CPUs, and it enables the object programmer to use condition variables for coordination between multiple invocations. In existing algorithms, a thread may only start or resume if there are no potentially nondeterministic conflicts with other running threads. A decision only based on past actions, without knowledge of future behaviour, must use a pessimistic strategy that can cause unnecessary restrictions to concurrency. Using a priori knowledge about future actions of a thread allows increasing the concurrency. We propose static code analysis as a way for predicting the lock acquisitions of object methods.

1 Motivation

Remote method invocation has become a standard approach for software that is both object oriented and distributed. The distributed object paradigm is an integral part of popular middleware systems such as CORBA, Java RMI, and .NET Remoting. Fault-tolerance extensions to these middleware systems support groups of replicated objects in addition to single remote objects [4, 5, 7, 8]. The replication infrastructure has the task to ensure that all replicas have an identical state at the same logical points of time. For object replication it is common to distinguish between active replication and passive replication. Passive replication uses a primary-backup scheme, in which a single primary replica executes all requests and updates the state of the backup replicas. In active replication, each replica executes all requests, and the execution must have a deterministic effect in order to ensure replica consistency. If multithreading shall be used, a scheduling algorithm that ensures determinism is mandatory. In passive replication, many systems update the state of backup replicas only after multiple modifications. State modifications not yet propagated to the backup replicas can be applied to them by re-executing method invocations from a request log. Such re-executions are consistent to the state of a failed primary only if a deterministic scheduling strategy is used. Because of this, deterministic scheduling is relevant for multithreading in both replication styles.

Most object replication systems execute all methods sequentially in total order. This strategy eliminates any nondeterminism that concurrent updates of the object state can cause. This approach, however, does not make use of modern multi-cpu or multi-core machines and thus results in an unnecessarily inefficient execution without parallelism. In addition this execution model is deadlock prone, e.g. when a remote object A while executing a method m_{A1} calls another remote object that in turn calls A again.

Optimistic update strategies, as widely used in database replication, are less convenient for distributed objects because of two main reasons. First, optimism only works well for large data sets in combination with transactions that access only small subsets thereof. This access pattern is in general not true for distributed objects. Second, optimism requires a rollback mechanism to undo wrong updates. This is hard to realise in typical environments for distributed objects without changing the compiler, the execution environment, or the application.

In previous work [11] we have surveyed programming

*This work has been partially supported by the FP6 Integrated Project XtreemOS funded by the European Commission (Contract IST2006-0033576)

1-4244-0910-1/07/$20.00 ©2007 IEEE.

models and algorithms that allow parallelism while keeping the determinism. The conclusion is that multithreading is the most flexible and powerful solution. Yet, multithreading bears the risk of non-determinism and makes necessary a sophisticated scheduling algorithm. Multiple deterministic scheduling algorithms [1, 2, 3, 11] have been proposed in the past. We included all of them in *FTflex* [12] the replication framework for our Java-based and CORBA-compatible Aspectix middleware. In the first part of this paper we give a survey on existing algorithms. All of them have in common that a thread may only start or resume if there are no potentially nondeterministic conflicts with other running threads.

A decision exclusively based on past actions, without knowledge of future behaviour, must use a pessimistic strategy that can cause unnecessary restrictions to concurrency. Using a priori knowledge about future actions of a thread allows increasing the concurrency. We propose static code analysis as a way for predicting the lock acquisitions of object methods. In the second part of this paper we present which information can be obtained by static code analysis and how it can help the algorithms to become more efficiently.

The rest of this paper is structured as follows. The next section gives an overview of the system model we use. In Section 3 we present all multithreading algorithms our replication framework contains. This is followed by a discussion on what information code analysis can generate and how the information can be provided to algorithms. Finally, we will conclude with a discussion on future work.

2 System Model

We assume that objects are replicated with a Java-based object middleware. This middleware shall be executed using a standard operating system and standard JVM. No internal modifications are made to these standard components. This means that the concurrent execution of multiple threads cannot be made deterministic by low-level means. Instead, we aim at deterministic scheduling at the application level, supported by the middleware infrastructure.

Such an approach requires that a high-level scheduling model be able to control the scheduling of threads. Thus, all relevant actions have to be intercepted by the application-level scheduler. We assume that the synchronisation uses binary, reentrant mutexes and condition variables for interaction between threads.

Even an entirely deterministic scheduler cannot guarantee determinism when the programmer uses non-deterministic functions or methods in the object implementation to be replicated. Such sources of non-determinism are often related to time or random numbers. In the case of Java, calls to a non-overwritten `hashCode` method are non-deterministic, too.

In a remote method invocation set-up, a request can only trigger the execution of a limited number of methods representing the remote object's public interface. We will refer to these methods as *start methods*.

As a prototype, we use our FT*flex* replication infrastructure of the CORBA-based Aspectix middleware. This replication infrastructure supports multithreading with the strategies described in the next section. In Java synchronisation there is a $1:1$ relationship between mutexes and condition variables. A condition variable can only be used when it has already been locked by a `synchronized` statement. Thus, in the following the terms *mutex* and *condition variables* only point out the different operations on the same object. That is `lock`/`unlock` in the first and `wait`/`notify` in the second case.

For intercepting synchronisation operations in the replica implementations, this infrastructure uses source-code transformations. All `synchronized` statements and all operations on condition variables are replaced with invocations at a scheduling module of FT*flex*. `Synchronized` is replaced with explicit `lock` and `unlock` calls. Because of that, referring to `lock` and `unlock` throughout the text is equivalent to referring to the beginning and the ending of a `synchronized` block. The mutex to be locked in a `synchronized` block is referred to as synchronisation parameter.

FT*flex* uses a group communication system [10] to guarantee that each replica receives all messages in a total order. Additional replication logic that is transparent to the client ensures a unique message identifier for each client request enabling replicas to ignore duplicated requests.

Finally, we assume that all access to shared object state is properly synchronised with native Java mechanisms. This holds for both critical read and write operations. Currently, our prototype only supports the standard synchronisation models, which uses `synchronized` methods and blocks. The extended synchronisation features of JDK 5, as provided in the `java.util.concurrent` package, are currently not supported. Our code transformation tool could, however, be extended to cover these extensions as well.

Throughout the text the term *nested invocation* occurs. That is the remote object calls an external service during the execution of a remote method. Nested invocations are non-trivial to handle, if the calling object is replicated, because each replica uses the same implementation and consequently will call the service. Because different return values of those calls are a potential source of non-determinism, we allow only one replica to do the call. The same replica spreads the reply to all other replicas.

3 Existing Approaches without Prediction

To the best of our knowledge, four algorithms for deterministic multithreading have been published so far. In our FT*flex* replication infrastructure, we implemented an adapted version of each algorithm as a basis for comparative evaluation. In addition to that, the infrastructure can be configured for sequential request execution. In the following, we describe the four algorithms called SAT, LSA, PDS, and MAT.

3.1 SAT

The SAT (*single active thread*) algorithm was first proposed by Jiménez-Peris et al. [6] for a conversational interaction with transactional replicas. Zhao et al. [13] have adapted the algorithm for object replication in the Eternal system. The FT*flex* variant of SAT [3] adds support for condition variables. The SAT algorithm does not use multithreading in the sense of concurrency. It allows a new thread to start or resume if a previously executing thread suspends, instead of waiting until final thread termination. A thread can either suspend by calling `wait` on a condition variable, or by starting a nested invocation. In both cases, another thread becomes active. If the reason for the suspension is not valid anymore (i.e., the nested invocation returns, or the `wait` operation is notified or times out), the thread is inserted in a queue. The thread at the head of the queue is removed from the queue and resumed only after the previously active thread suspends or terminates.

The SAT algorithm improves the sequential execution model, as it uses the idle time during nested invocations for executing new requests. It avoids potential deadlocks that arise if chains of nested invocations loop back to an object that waits for the nested invocation reply. In addition, our SAT variant allows using standard Java synchronisation and condition variables. Nevertheless, it does not make use of multiple physical CPUs as there is only one running thread at a time.

3.2 LSA

The LSA (*loose synchronisation algorithm*) was originally published by Basile et al. [2] and uses a simple leader-follower scheme. It is the only algorithm that depends on frequent inter-replica communication. A single replica acts as the leader and has no restrictions regarding the execution order. All other replicas are bound to the decision that the first one makes. This algorithm was published without support for condition variables. As the leader replica may take any arbitrary decision and condition variables have to be locked before they can be used, inserting support for them in the in FT*flex* version of the algorithm did not raise major problems.

3.3 PDS

In the PDS (*preemptive deterministic scheduling*)algorithm [1] a pool with a fixed number of threads is used. For each request arriving at the system a thread starts running. It may run until it requests its first lock. The lock is granted only when a sufficient number of other threads have also requested their first lock. At this point it is possible to determine if conflicts occur and to solve them. After all threads have processed the critical section all of them are allowed to run to their next lock request. In an optimised version each thread is allowed to request two locks. Yet, this does not dispose the weaknesses of the PDS algorithm. Lock acquisition does not start before a significant number of requests have arrived and requested a lock and the algorithm expects all requests to trigger method calls with similar profile

As the algorithm as described so far allows starvation of threads, the FT*flex* implementation generates dummy messages to fill the request queue. Additional messages guarantee that all requests are eventually processed, even if no new external messages arrive. The price to pay is higher communication overhead, as all dummy messages must pass the group communication system. Adding support for condition variables turned out to be even more complicated due to the fact that the algorithm is not prepared for a thread blocking in a `wait` operation and only waking up by intervention of a thread that maybe has not even arrived yet.

3.4 MAT

The MAT (*multiple active threads*) algorithm [11] is an extension to the SAT algorithm that supports multiple, concurrently running threads (active threads) at a time. Active threads comprise one primary active thread and multiple secondary active threads. Besides these, there is a queue with blocked threads. Primary and secondary threads differ in the operations they are allowed to execute. The primary thread may request locks, secondary threads may not. If a secondary thread requests a lock, it is blocked and is continued only after it has become primary. The oldest secondary thread becomes primary when the current primary blocks, finishes, or issues a nested invocation and no blocked primary can continue running.

The MAT algorithm allows real multithreading with multiple active threads. This is especially useful for threads that never request any locks, because they can run without interruption; but also for threads that issue computations before changing the object state. Nevertheless, a secondary thread that requests a lock is blocked until it is primary, no

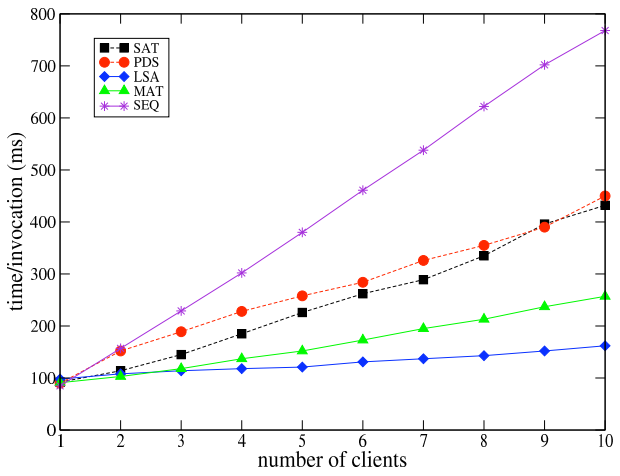


Figure 1. Benchmark for multiple deterministic scheduling algorithms

matter whether the lock that itself and the current primary will request conflict or not. Even worse, the algorithm does not recognise when a thread has requested and released all of its locks and will never request one again. Such a thread cannot profit from being primary and delays the execution of secondary threads waiting for a lock.

3.5 Comparison of SAT, LSA, PDS, and MAT

All of the algorithms presented above were subject to extensive benchmarking [9] leading to the result is that there is no single best algorithm, but for all of them exist scenarios in which they outperform all others. Figure 1 presents the results for one of the benchmarks. It shows the average time it takes to process a remote method invocation as a function of the number of clients in a set-up with three replicas for all four algorithms and an additional SEQ algorithm using sequential request processing. Both clients and replicas were located in the same local area network. All clients called the same method. The implementation of that method in the remote object does ten iterations of a loop. Each iteration performs the following operations:

- with probability 0.2, simulate a nested invocation (duration approx. 12 ms)
- with probability 0.2, simulate a local computation (duration 10 . . . 20 ms)
- execute a sequence of lock, state update, unlock, using a mutex chosen by random from a set of 100 mutexes.

The random selection of the nested invocation and the local computation was used to simulate different kinds of object behaviour. The rather big number of mutexes is to simulate fine-grained locking of the object state. To guarantee deterministic behaviour the clients were responsible for all random decisions and passed them as method parameters.

As it might have been expected, the SEQ algorithm scales worst, as it does not make use of the idle time during nested invocations. PDS and LSA scale far better than SEQ, but also far worse than MAT. LSA scales best, but this is mainly caused by the fact that the leader may decide without restrictions and the client only waits for the first reply. Thus, the response is directly sent to the client without waiting for other replicas. Nevertheless, this algorithm poses a high load on the network caused by the need for frequent broadcast communication and thus may behave worse in WAN setups. Furthermore, it depends on the leader replica to make decisions. In case of a failure this might lead to a high take-over time that does not exist for MAT and the other algorithms, as they treat all replicas equally.

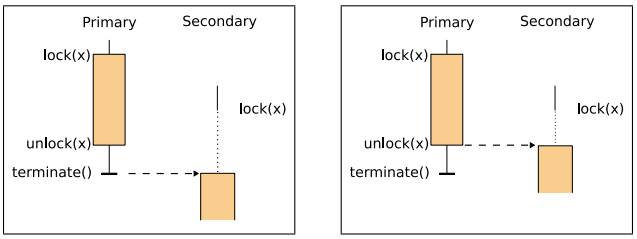
4 Code Analysis

The pessimism all algorithms use is very restrictive when very fine grained locking is used, i.e. there are many mutexes and there are methods that use disjunctive mutex sets. In order for the algorithms to be able to decrease the pessimism in their decisions they must have information about each thread's future locks. For gathering this information we propose static code analysis.

For our environment we take the assumptions already presented in Section 2. Furthermore, we will use additional, very restrictive assumptions that we try to relax later on:

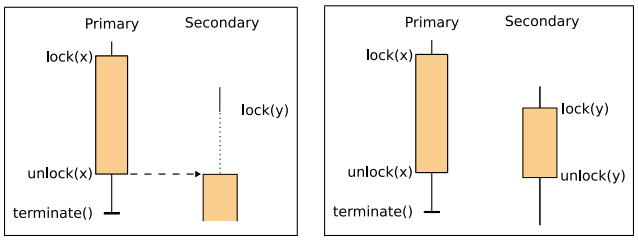
- There are no `synchronized` statements within loops; that is `for`, `while` and `do` . . . `while`.
- All methods that are called are `final`.
- There is no recursion, so there is only a limited number of paths the execution may take for each start method.

The FT*flex* deployment includes a code transformation process, just before the final compilation, which replaces `synchronized` statements with calls to the scheduler. The transformation is done by our Transformation Process Language (TPL). TPL is our toolbox language for detailed transformations on multi-language source code models. So called *processes* conduct queries and transformation steps at the syntax tree backend of the model elements. Based on a traverser interface, which is common to all supported languages, the processes navigate on syntactical and semantical relationships between elements. Complex transformations are composed of path-structured model queries, list-based iterative expressions, and calls to operators and further processes. New structures can be created manually, element by element, or by passing annotated text fragments



(a) MAT (b) Ideal

Figure 2. Locking pattern after releasing the last lock



(a) With Last Lock Analysis (b) Ideal

Figure 3. Locking pattern for non-conflicting mutexes

to the parser of the respective target model. A LISP-style concrete syntax reflects the hierarchical structure of the target models and allows us to form complex multi-valued expressions. In addition to code transformation the TPL also supports adding new code; code injection.

Code analysis is an important step towards increased concurrency, but not the final one. A further step is to give the scheduler access to the information from the analysis. We solve this in a twofold manner. First, the analysis generates static information that is used to initialise the scheduler. Second, we insert additional code in the object implementation telling the scheduler at runtime how to manage the static information on a per thread basis.

The optimisations a scheduler can make in comparison to pessimistic schedulers depend on the scheduling algorithm and the information provided by the code analysis. In the following we first describe which information is mandatory for that the scheduler recognises when a thread's last lock has been requested and show how the current MAT algorithm has to be changed to make use of this information. Then, we show how to predict a thread's future locks. We sketch in short how the current architecture may be changed so that both information aware and information unaware algorithms are supported. We also sketch how the MAT algorithm can be chaged to support lock prediction. However, we do not present a totally worked out and proven version of this algorithm, as it is still work in progress and subject to future investigations. Finally, we propose how to relax above restrictions.

4.1 Last Lock Analysis

In the MAT algorithm a secondary becomes primary only after the current primary has terminated or suspended (Figure 2(a)). Usually, the last unlock is followed by a final computation. In the case of FT*flex* the thread builds the reply message that is sent back to the client. The final computation has no influence on the determinism of mutex locking. Providing the scheduler with information about when a thread's last lock has been released enables to change the primary even before thread termination (Figure 2(b)).

One approach to finding the last lock is to provide a list in the scheduler containing the locks that will be requested by one thread. To be able to provide the scheduler with the information of locks, a list of all of `synchronized` blocks the programme flow can pass is necessary. Each of them gets a globally unique *syncID*. Now, by code analysis, we can figure out all execution paths for all start methods and the syncIDs of the `synchronized` blocks on the paths. Thus, we get a list of syncIDs for each start method and with it all the static information the scheduler needs.

The scheduler is initialised with that information at start-up. For each thread that is started, a local copy of the static information concerning the thread's start method is made and attached to the thread descriptor in the scheduler. Changing the scheduler interface to `scheduler.lock(syncID, mutex)` allows the scheduler to correlate lock requests for mutexes with the syncID table and manipulate the latter accordingly (bookkeeping).

The scheduler's bookkeeping does only work correctly, when it gets all information available. That is information about syncIDs locks are requested and released for, but also on syncIDs, that are ignored, because a path is used they are not on. Injecting `scheduler.ignore(syncID)` on all paths without a lock call for syncID provides the necessary information to the scheduler.

4.2 Lock Parameter Analysis

Finding the last lock provides information for tuning the algorithm for requests that issue their last lock before a final computation. However, concurrently processed requests that have non-overlapping future lock requests do not benefit from this information as much as they could. Figure 3(a) shows an example for such a situation. The primary thread requests and releases a lock on mutex x and finishes afterwards. The first secondary thread requests a lock for

mutex y, but has to wait until the primary has released x. In an ideal case the scheduler, like in Figure 3(b), would recognise that x is the primary's last lock, that there is no relationship between x and y, and would grant the lock to the secondary.

For enabling more concurrency while still keeping determinism, the algorithm must be able to check, if both threads overlap for the current lock request and for all future lock requests. This means - assuming an order on the threads - a thread will be allowed to acquire the lock only if it can be granted that all threads that are better with respect to the ordering will not request a lock on the same mutex in the future. Thus, we aim at gathering information about future locks; the sooner all of them are known the more concurrency may be achieved.

In order to determine which objects will be locked during method execution, we need to inspect the `synchronized` parameter and find out when this parameter is assigned the last time. This is straight forward, if the parameter is `this`, a method parameter, or an object that is local to the method. In the first case the parameter is final, in the latter ones we can find out about the last assignment by code analysis. It is impossible to determine the last assignment for other parameters, like an instance variable, a globally accessible object, or the return value of a method call. We will refer to those as *spontaneous parameters*, because the parameter is unknown until the locking happens. There is no need to generate additional static information for lock parameter analysis compared to last lock analysis. However, injecting additional code is not preventable.

For that the scheduler is able to benefit from the analysis, we insert the following call to the scheduler, right after the last assignment `scheduler.lockInfo(syncID, mutex)`. It marks the appropriate entry in the thread's list of syncID. In the following, we say a thread is *predicted*, if all entries in the list are marked.

In case of spontaneous parameters, we do not insert any additional instructions. Locking such a mutex is treated like a call to `lockInfo` followed by a call to `lock`. Threads executing methods containing locks with spontaneous parameters get only predicted when all of them have been passed.

The left side of Figure 4 shows a method of a class implementation that is subject to code analysis and code transformation. The right side shows the outcome of this process omitting error handling code. The changes to traditional FT*flex* code transformation are highlighted. The method shown has two paths with one `synchronized` statement in each of them. The first one has a non-spontaneous parameter passed as method parameter that is not changed while processing the method, i.e. it is announced right after the method start. The second one has a spontaneous parameter that cannot be announced. The example contains all cases of code injection presented so far.

4.3 Extending the FT*flex* Architecture and the MAT Algorithm

The code analysis, source code injection, and bookkeeping described in the last two subsections are orthogonal to any scheduler implementation, as the goal is to provide information to the scheduler, but not to lay down the way the scheduler has to use them. Thus, in a future architecture we envisage the scheduler to be built up of two modules; a bookkeeping module and a decision module. The bookkeeping module contains all static and thread-wise information, reflecting the knowledge about the threads' current and future lock acquisitions. Calls from the object implementation always reach the bookkeeping module first, triggering changes to the syncID tables. Then, they are passed on to the decision module.

The bookkeeping module also offers an interface to the decision module the scheduler implementation may use to find out about conflicting locks. The decision module may use the bookkeeping module, but does not have to. This enables the re-use of existing scheduler implementations as decision modules and the implementation of new schedulers without having to re-implement the bookkeeping module for each scheduling algorithm.

As a starting point for a new scheduler implementation we use the MAT algorithm, as we consider it to be the most flexible one. Instead of only using one active primary thread, we aim at a queue of active threads that are in principle equal. A thread t only gets a lock when all threads preceding it in the queue are already predicted and none of them conflicts with the lock requested by t. Otherwise t is suspended. The algorithm must check suspended threads when certain events happen.

When t is waiting for a predicted thread these events are:

- A thread conflicting with t releases the mutex t is waiting for,
- A thread conflicting with t is removed from the list,

Let t_u be the first thread in the list that is not predicted and t be a successor of t_u, then the events are

- t_u is removed from the list.
- t_u becomes predicted.

A completely worked out extension to the MAT algorithm is still subject to further investigations, we have not been able to figure out yet how the algorithm should proceed when a thread calls `wait` or does a nested invocation.

```
private Object myO;

public void foo(Object o){

 if (myO.equals(o))
  synchronized(o) { }

 else
  synchronized(myO){ }
}
```

⟹

```
private Object myO;

public void foo(Object o){
 scheduler.lockInfo(1, o);

 if (myO.equals(o)){
  scheduler.ignore(2);

  scheduler.lock( 1, o);
  scheduler.unlock( 1, o);
 }
 else {
  scheduler.ignore(1);

  scheduler.lock( 2, myO);
  scheduler.unlock( 2, myO);
 }
}
```

Figure 4. Code transformation and injection by the code analysis process

4.4 Relaxing Previous Assumptions

The code analysis presented so far is restricted by the assumptions from the beginning of this section. As they limit the possibilities a programmer has, weakening them seems desirable for us. Thus, we will sketch how this may be done.

Up to now, we assumed that no `synchronized` statements are used within loops. We distinguish loops and general recursion, because loops can be handled more easily: They are local to a method, easy to detect and less powerful than arbitrary recursion. Without locking within loops, it is sufficient for the scheduler to maintain a list of syncIDs.

Locking within loops destroys knowledge about the locking quantity. So far, an element in the syncID list could only be requested once or not at all. Loops introduce an uncertainity, because not only the quantity is unknown, but also the mutex might change for each loop, so we do not know a priori how many different mutexes will be requested at all.

The occurrence of locks within loops has no influence on a new scheduling algorithm. It does however, influence the static information to be generated and also the bookkeeping module.

Our approach to handling locking within loops is to distinguish two different kinds of loops: The first kind locks the same mutex in each loop. It exists, when the `lock` parameter is not spontaneous, is assigned before the loop, and is not assigned within the loop. Otherwise, the mutex can change in each loop. So, for the first kind, the announcement process for future locks does not change. However, the mutex must be respected as long as the loop has not been finished and not until the `unlock`. In the second case, in general, we can neither tell how often the loop is processed nor how many mutexes will be locked nor which mutexes this will be. That means, we cannot consider all mutexes to be known before the loop finishes. Accordingly, the thread will only be predicted after having passed all such loops.

Our second restriction is that all methods called have to be `final`. The reason therefore is that only in this case we can be sure that the method associated with the static type is really the method that will be called at runtime. Giving up this assumption will not be possible without developer support. One possibility is that the developer must assure that the static type is always equal to the runtime type. Another approach is to use a repository with a certain number of classes and information about the locking they use. At runtime the runtime type can be checked and the appropriate information be generated from the repository. The efficiency of this approach depends on how often the runtime type changes. The risk is that bookkeeping eats up all performance gained by increased concurrency.

The third restriction is that no recursion occurs. Our approaches to handle this is to either to step back to a simpler algorithm or to build up additional path information before each single method call. As this will probably produce high runtime overhead, we currently favour the first approach. All of the topics are also subject to future work.

5 Conclusion and Future Work

In this document we presented why object replication benefits from deterministic multithreading algorithms. We gave a survey on existing deterministic multithreading schedulers and an overview of our replication framework. Our position is that concurrency can be increased by providing additional information about future lock acquisition to the scheduler. Static code analysis can gather this information. We presented two levels of granularity the information can have; one for recognising a thread's last lock and the other for lock prediction. Then, we sketched changes to our current environment and how to change the existing MAT algorithm for that it can handle the additional information. Finally, we showed up how to weaken the restrictions we made for code analysis.

For the future we are targeting the implementation and formal verification of an extended MAT algorithm using information from static code analysis. Besides the optimisations discussed so far, we are planning other extensions to our multithread scheduling system. First, we would like to add support for the Java 1.5 concurrency specification. Furthermore, we think of sophisticated data flow analysis that may help to statically determine which threads will never interfere at all. Moreover this can also help to determine upper bounds for loops. Additionally we are planning to develop a request analyser that chooses the appropriate scheduler at runtime depending on the client interaction patterns and the methods' lock pattern. Finally, we want to analyse to what extent static information helps to optimise performance and at which point performance decreases again due to runtime overhead; this may be done by providing a mathematical model for locks, methods and client interaction.

References

[1] C. Basile, Z. Kalbarczyk, and R. Iyer. Preemptive deterministic scheduling algorithm for multithreaded replicas. In *Proc. Int'l Conf. on Dependable Systems and Networks (DSN), 2003.*, 2003.

[2] C. Basile, K. Whisnant, Z. Kalbarczyk, and R. Iyer. Loose synchronization of multithreaded replicas. In *SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, page 250, Washington, DC, USA, 2002. IEEE Computer Society.

[3] J. Domaschka, F. J. Hauck, H. P. Reiser, and R. Kapitza. Deterministic multithreading for Java-based replicated objects. In *Proc. of the 18th IASTED Int. Conf. on Parallel and Distributed Computing and Systems (PDCS'06, Dalles, Texas, Nov 13-15, 2006)*, 2006.

[4] P. Felber and P. Narasimhan. Experiences, strategies, and challenges in building fault-tolerant CORBA systems. *IEEE Trans. Comput.*, 53(5):497–511, 2004.

[5] R. Friedman and E. Hadad. FTS: A high-performance CORBA fault-tolerance service. In *Proceedings of the Seventh International Workshop on Object-Oriented Real-Time Dependable Systems*, 2002.

[6] R. Jiménez-Peris, M. Patiño-Martínez, and S. Arévalo. Deterministic scheduling for transactional multithreaded replicas. In *SRDS '00: Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, page 164, Washington, DC, USA, 2000. IEEE Computer Society.

[7] S. Maffeis. Adding group communication and fault-tolerance to CORBA. In *Proceedings of the Conference on Object-Oriented Technologies, (Monterey, CA), USENIX*, pages 135–146, 1995.

[8] N. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Transparent consistent replication of Java RMI objects. In *DOA*, pages 17–26, 2000.

[9] H. P. Reiser. *Flexible and Reconfigurable Support for Fault-Tolerant Object Replication*. PhD thesis, Fakultät für Ingenieurwissenschaften und Informatik, Universität Ulm, 2006.

[10] H. P. Reiser, U. Bartlang, and F. J. Hauck. A reconfigurable system architecture for consensus-based group communication. In *Proc. of the 17th IASTED Int. Conf on Parallel and Distributed Computing and Systems (Phoenix, AZ, USA, Nov 14-16, 2005)*, 2005.

[11] H. P. Reiser, F. J. Hauck, J. Domaschka, R. Kapitza, and W. Schröder-Preikschat. Consistent replication of multi-threaded distributed objects. In *SRDS '06: Proceedings of the 25st IEEE Symposium on Reliable Distributed Systems (SRDS'06)*, 2006.

[12] H. P. Reiser, R. Kapitza, J. Domaschka, and F. J. Hauck. Fault-tolerant replication based on fragmented objects. In *Proc. of the 6th IFIP WG 6.1 Int. Conf. on Distributed Applications and Interoperabel Systems - DAIS 2006 (Bologna, Italy, June 14-16, 2006), LNCS 4025*, pages 256–271, 2006.

[13] W. Zhao, L. E. Moser, and P. M. Melliar-Smith. Deterministic scheduling for multithreaded replicas. In *WORDS '05: Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 74–81, Washington, DC, USA, 2005. IEEE Computer Society.