Weaving Atomicity Through Dynamic Dependence Tracking

Suresh Jagananthan
Department of Computer Science
Purdue University
West Lafayette, IN 47907
suresh@cs.purdue.edu

[1]

Abstract

Programmability is the key hurdle towards effectively utilizing next-generation high-performance computing systems. Current trends in CMP processor design point to the emergence of many-core architectures, in which a single chip will support tens to potentially hundreds of cores. Systems constructed by aggregating these processors can enable parallel execution of thousands of threads.

Transactional memory *(TM) has been the subject of significant interest in both academia and industry because it offers a compelling alternative to existing concurrency control abstractions, making it especially well-suited for programming applications on scalable multi-core platforms. TM abstractions permit logically concurrent access to shared regions of code, but ensure through some combination of hardware, compiler, and runtime support that such accesses do not violate intended serializability invariants. By doing so, transaction-based abstractions eliminate pernicious errors such as data races that can easily occur using locks, without compromising performance.*

While the atomicity and isolation guarantees provided by transactions lead to greater composability and modularity than available using locks, these guarantees may require severe constraints on programmability. In this paper, we describe compiler and runtime techniques that allow structured communication among atomic regions to take place, thus selectively relaxing isolation invariants. Unlike existing proposals, our techniques are completely transparent, and provide a rational semantics for the interplay between transactions, message-passing abstractions, and exceptions.

[1]This material is based on work supported by the National Science Foundation under Grant No. CSR-AES 050937.

1 Introduction

Next-generation microprocessors built on multi-core processor designs promise the availability of commodity scalable parallel systems on the desktop. Effectively and safely programming these systems is a subject of much recent interest. The language community has developed new abstractions to safely extract concurrency from programs without increasing coding complexity. Atomic or transactional regions are a simple method for specifying sections of code which may have potential conflicts at runtime. The underlying compiler or runtime system enforces safety guarantees on such regions of code, removing the burden from the programmer.

Transactional memory has been the subject of significant interest in both academia and industry because it offers a compelling alternative to existing concurrency control abstractions, making it especially well-suited for programming applications on scalable platforms. Because transactional memory implementations often support an optimistic concurrency model, they can be used to safely allows speculative access to data by a large number of processors without requiring global program analysis. TM abstractions permit logically concurrent access to shared regions of code, but ensure through some combination of hardware, compiler, and runtime support that such accesses do not violate intended serializability invariants.

Atomic regions improve composability over locks. For instance, if two pieces of code using locks are combined (e.g., through function calls), the resulting behavior can be undesirable (e.g., deadlocks, data races, etc.) unless all details about the locks are used is known (e.g., in which order are the locks acquired and released, what data do they protect, etc.). This lack of composability severly degrades programmability. The problem is exacerbated for programs intended to execute on highly-parallel architectures which may support hundreds of threads accessing large complex shared data structures.

Transactions improve composability by ensuring that the

read and write accesses of one transaction do not conflict with the operations provide by another concurrently executing transaction. When such conflicts are detected, one of the transactions is aborted, and required to re-execute. By making state changes globally visible only when a transaction successfully commits, programmers can reason using a simple execution model, in which transactions execute in a serial, one-at-a-time order.

Serializability, through atomicity and isolation constraints, is the key underlying property for any transactional semantics. Unfortunately, many programs require threads to communicate in ways that inherently violate isolation, the property that gives at most one transaction executes at a time. For instance, consider two transactions involved in a two-way communication where one produces data that is to be consumed by the other, which in turn produces data (or sends a signal) to be consumed by the first. The existence of such communication may be hidden by several layers of abstractions, and may not even be apparent at the point where the transactions are defined.

Current solutions cannot guarantee correct behavior for these programs. Most implementations that uncompromisingly enforce isolation would cause transactions to defer deliver of data until commit-time, forcing neither transaction to make progress. What is in fact needed is the ability to *selectively* relax isolation by allowing actions performed within the transaction to be made globally visible before the transaction commits. The net effect of this relaxation is to *weave* a larger atomic scope from a collection of smaller ones. If a transaction T_1 produces data that is consumed by another transaction T_2 and T_2 subsequently aborts, any of T_1's actions that may have implicitly depended on T_2's consumption may not be reverted as well. In other words, the effect of the communication is tantamount to the dynamic construction of a larger atomic region comprising both T_1 and T_2.

In this paper, we describe compiler and runtime techniques that can be used to reconcile notions of atomicity and isolation with inherently non-isolated actions such as message-based communication, signals, and exceptions.

2 Motivation

Our design is motivated by issues of composability and programmability. Composability is difficult to achieve in concurrent programs especially in the presence of errors, synchronizaiton, or exceptions. Suppose a thread t_1 intended to execute atomically propagates data to thread t_2. If t_1 raises an exception or encounters an error prior to completion, t_2's execution is indirectly affected, and may allow t_2 to see stale or inconsistent values. To deal with such possibilities, exceptional conditions and errors must be coordinated between both t_1 and t_2. Modularity and abstraction is thus negatively impacted.

Current research in software transactions[6, 7, 8, 17, 23, 21] address some of these issues through the use of open-nesting [11, 14]. Here, the effects of an open (inner) nested transaction are made globally visible on commit, even if the outer (parent) transaction has not yet committed. This semantics is in contrast to the semantics of *closed* nested transactions in which the effects of a nested transaction are only made visible once the top-most enclosing transaction commits. Open nesting permits relaxed isolation. However, if the parent transaction aborts, atomicity properties are violated: the effect of the inner transaction has already been visible to computations outside the transaction. To remedy the violation, a *compensation* action can be associated with each open-nested transaction. These user-defined actions are executed upon abort of the parent, and are intended to undo (semantically) the global effects of the nested transaction. Unfortunately, because compensations are user-defined, there is no guarantee that the compensation will in fact undo all necessary effects, or even feasible.

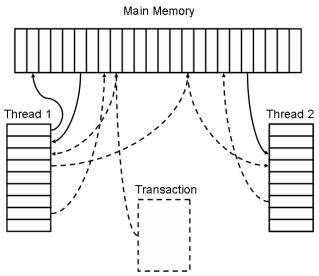
Our approach can be viewed as a middle-ground between the restrictive programming model of closed-nested transactions and the overly general model supported by open-nested transactions. We are interested in exploring the extent to which compiler and runtime techniques can provide relaxed isolation. While we concede that in certain cases, semantic knowledge about a transaction's effects are necessary in order to validate isolation and atomicity properties, we believe that in most instances, efficient dynamic dependency tracking techniques can provide the benefits of relaxed isolation without imposing any burden on programmability.

3 Programming Model

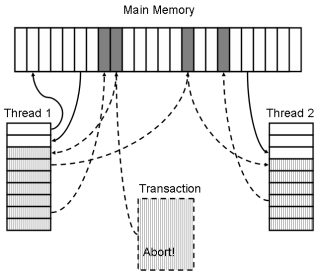
We consider a programming model in which operations on shared data are protected within an atomic region. Besides reads and writes to shared memory, a thread may induce effects via a communication or signal, raise an exception, or spawn other threads within the dynamic context of an atomic region. For the purposes of exposition, we consider data transmitted via a communication event (e.g., sends or receives on a channel) as being *non-isolated*; all other data values read or written by a transaction are considered *isolated*. The receipt of non-isolated data by one transaction produced by another results in a dependency between the two transactions. If the producer aborts due to a serializability violation of its isolated data, all dependent transactions established as a result of communication of non-isolated data must also abort. If the recipient of non-isolated data aborts, the producer is also obligated to abort, although it may be possible to limit the extent of the abort to the point immediately prior to the communication that

established the dependency. Observe that before the communication was established, the producer had executed in isolation.

While our design generalizes the behavior of transactions (i.e., communication within transactions is supported), the underlying implementation is more complicated because of the need to (a) effectively track dependencies among transactions, and (b) provide mechanism to revert potentially many concurrently executing dependent transactions to preserve global atomicity properties.



(a)



(b)

Figure 1. (a) Transactions can interact in non-trivial ways through the transmission of non-isolated data. (b) When a transaction aborts, the state of all threads that have witnessed its effects must also be reverted.

To illustrate these ideas, consider Fig. 1(a) that shows the interaction of a collection of transactions. Transaction initiates a non-isolated communication action that is witnessed by Transaction 1. Transaction 1 subsequently initiates communication that is seen by transaction 3. Further communication between 1 and 3 occur before transaction 3 aborts. At this point, all of transaction 3's dependent actions executed by other transactions must also be aborted (see Fig. 1(b)). The shaded blocks in memory show the locations affected, and the gray boxes in the transactions highlight the computations that must be reverted. It is clear that transactions 1 and 2 can no longer attempt to commit because of the abort of transaction 3. Whether they are forced to abort completely, i.e., whether the transactions must be re-executed in their entirety, or whether they need only be reverted to the point of the initial communication which established a dependency is an artifact of the implementation. Our current implementation will in fact avoid reverting the entire transaction.

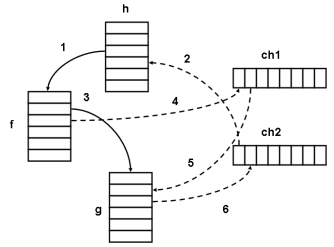
4 Motivating Example

```
fun h() =
  let fun g() =
        ...;  x = recv(ch1)[5];
        ...;  send(ch2,x+1)[6]; ...
      fun f(v) =
         spawn(g())[3]
         in ...;
            send(ch1,v)[4];
            ...;
         end
  in ...;
     spawn(f(v) handle Retry => f(v'));)[1]
     ...; y = recv(ch2)[2]; ...
  end
spawn(h())
```

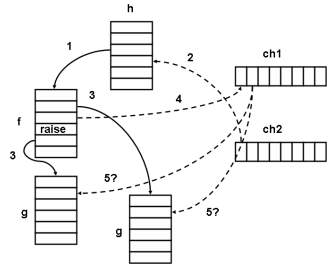
Figure 2. Concurrent programs compose poorly and violate atomicity in the presence of exceptions. The super scripts represent transition edges given in Fig. 3

To further motivate the need for relaxed isolation, consider the example shown in Fig. 4. In this program, the thread evaluating function `h` spawns a new thread to evaluate `f(v)`. Function `f` in turn spawns a thread to evaluate `g`. In addition to these function calls, these three threads communicate with one another using synchronous message passing. For example, `f` sends value `v` on channel `ch1`, and blocks until `g` reads that value (`recv(ch1)`). Similarly, `g` sends a value (the result of `x+1`, where `x` is bound to the value previously read from `ch1`) along channel `ch2`. The outermost thread computing `h` synchronizes on this channel and reads the value deposited by `g`. When all goes well, the value `y` in `h` is simply `v+1`. The control and data flow for this simple example are given in Fig. 3(a). Control flow is depicted as solid arrows and data flow as dashed. Threads are modeled as downward growing stacks and channels as a list of ordered list of values.

Issues become substantially more complicated in the presence of exceptions and errors. Suppose that in the course of evaluating `f(v)` an exception is thrown. If the exception is thrown before value `v` is deposited on channel `ch1`, both the threads evaluating `g()` and `h()` block. Simply terminating the thread spawned from within `f` is insufficient to unblock the outer thread. Therefore, the exception handler must pass some value to `g` through channel `ch1`. In this situation a simple re-execution of `f` would suffice.



(a)



(b)

Figure 3. Control and data flow when (a) no exception is raised, and (b) when an exception is raised after `f` sends value `v` on channel `ch1`.

Suppose the exception is thrown after the message is sent (see Fig.3(b). Since the exception handler simply retries the call (supplying a new argument `v'`), there now exists a race condition on channels `ch1` and `ch2` from two threads both computing `g()`. In this case it is unclear which value will eventually be received by `h()`. Only one value from the two threads executing `g()` will be read, resulting in the other producer thread blocking. Of course, the handler could attempt to terminate the thread computing `g()` before it retries, but the race condition still exists as observed by `h()` since the message send on channel `ch2` may or not have happened at the time the thread is terminated. If the send has not occurred, the receipt on channel `ch2` observes `v'+1`, and if it has, the receipt observes `v+1`.

To ensure that `f`'s effects are atomic with respect to `g` and `h` requires weaving a complex exception handling protocol through these procedures. What is really required is a mechanism that allows the `Retry` exception to effectively roll-back any read performed on channel `ch2` by the outer thread, since such reads depend on the publication of a stale value `v` from the inner thread. In the absence of such a mechanism, the actions performed by `f` in the presence of exceptions are no longer atomic. Remedying the atomicity failure thus requires not only revocation of all threads spawned within `f`, but also any effects on other threads that were propagated either directly by `f` or indirectly through other spawned threads.

```
fun h() =
    let fun g() = atomic (as before)
        fun f(v) =
             let as before
    end
  in ...;
     spawn( atomic (f(v))
            handleAbort Retry => f(v'))
     ...;)
  end
spawn( atomic(h()))
```

Figure 4. Ensuring atomicity using safe speculative threads.

Using transactions that support non-isolation, we can implement a composable and safe version of our example program.

The modifications to the original program are slight. Program regions that must be executed atomically are marked as such. The nested atomic region for the call `f(v)` is a nested transaction. If the `Retry` exception is raised during its evaluation, the current transaction is aborted, and a new transaction to evaluate `f(v')` is instantiated. Observe that these changes do not entail propagation of error handling protocols among the different transactions in the example. The implementation of ensures the restoration of a consistent global state that is untainted by any of `f`'s effects. In this example, this entails termination of the thread spawned within `f`, and resetting the control state for the thread computing `h` to be that immediately prior to the read on channel `ch2` (assuming earlier operations in `h` do not operate over channel `ch1` or `ch2`). From the perspective of the outer transaction there is no visible effect as a result of the revocation that occurs within the `Retry` handler in `f`. Composability and atomicity is thus maintained.

5 Implementation

Our implementation is incorporated within MLton [10], a whole-program optimizing compiler for Standard ML. The implementation is focused around a dynamic depen-

dency graph which tracks dependencies among transactions that exercise relaxed isolation. We utilize light-weight per-thread checkpointing to revert aborted speculations [25] and synthesize safe state based on the dependency graph. The main change to the underlying infrastructure is the insertion of write barriers to track updates to shared data. Per-thread checkpoints are established by capturing first=class continuations. State restoration is thus a combination of restoring continuations as well as reverting references. The implementation is roughly 2K lines of code to support our data structures, checkpointing, and restoration code.

All experiments are performed on an Intel P4 2.4 GHz machine with one GByte of memory running Gentoo Linux, compiled and executed using MLton release 20051202.

5.1 Dependence Graph

Ou implementation tracks how transactions are dependent on one another. We can model a transaction's execution as a sequence of nodes; each node stores a transaction's continuation. The initial node corresponds to the transaction's creation point, and subsequent nodes refer to points within its execution. The last node within the chain is the most recently monitored action for the transaction. We impose an ordering on nodes to define a global timeline. Each successive node within a transaction's chain is dependent on previous nodes within the chain. We define a transaction's *current node* as the last node within its execution chain. A communication action establishes an edge between two nodes belonging to different transactions.

The purpose of monitoring such inter-thread dependencies is to dynamically determine who has been affected by a transaction's execution. When a transaction is aborted, we traverse the graph through a depth first search starting from the current node of the transaction to identify non-isolated actions. Such a search will correctly discover all transactions (and the precise point in their execution) which depend on the values the aborting transaction has modified and/or created. This depedence graph thus enables a sequence of cascading aborts to be effected based on the dynamic communication behavior of the program.

5.2 Graph Representation and Optimizations

The main challenge in the implementation was developing a compact representation of the communication graph. We have implemented a number of node/edge compaction algorithms allowing for fast culling of redundant information. For instance, any two nodes that share a backedge can be collapsed into a single node. We also ensure that there is at most one edge between any pair of nodes. Any addition to the graph affects at most two transactions. We use transaction-local meta-data to find the most recent node for each transaction. The graph is thus never traversed in its entirety. Furthermore, we do not need to store the entire graph for the duration of program execution. As the program executes, parts of the graph will become unreachable. The graph is implemented using weak references to allow unreachable portions to be safely reclaimed by the garbage collector, thus greatly reducing memory overheads.

6 Case Study - Swerve

`Swerve` [10] (see Fig. 5) is an open-source third-party Web server wholly written in Concurrent ML (CML) [15]. The server is composed of five separate interacting modules. Communication between modules and threads makes extensive use of CML message passing semantics. Threads communicate over explicitly defined channels on which they can either send or receive values. We consider the interactions of three of `Swerve`'s modules: the `Listener`, the `File Processor`, and the `Timeout Manager`. The `Listener` module receives incoming `HTTP` requests and delegates file serving requirements to concurrently executing processing threads. For each new connection, a new listener is spawned; thus, each connection has one main governing entity. The `File Processor` module handles access to the underlying file system. Each file that will be hosted is read by a file processor thread that chunks the file and sends it via message-passing to the thread delegated by the `Listener` to host the file. Timeouts are processed by the `Timeout Manager` through the use of timed events on channels. Threads can poll these channels to check if there has been a timeout. In the case of a timeout, the channel will hold a flag signaling time has expired, and is empty otherwise.

Each request is processed through many communicating threads located in separate modules, and its completion depends on the timeout quantum. We can model this execution behavior through transactions which are managed by a timer resource. Transactions created by the `Listener` and `File Processor` are closed over a timer resource. If at the point of commit, the resource has expired, a serializability violation is observed, and the transaction is aborted. The web server is then ready to process another request, or alternatively re-process the same request. If the request completes, the timer is reset, and the transaction commits the data it has received. Utilizing transaction thus avoids explicit thread clean-up code when a timeout occurs. Such procedures break modularity and composability because they span multiple modules (see the dashed lines in Fig. 5). Note that relaxed isolation is critical to the implementation: while each module should execute transactionally, the complex interactions among them make it infeasible to assume a closed-nested transaction model. Data can be communicated by the `File Processor` to the

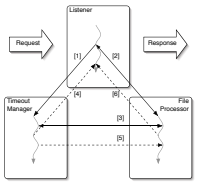


Figure 5. `Swerve` **module interactions for processing a request (solid lines) and error handling control and data flow (dashed lines) for timeouts. The number above the lines indicates the order in which communication actions occur.**

`Listener` provided that a timeout does not occur before all file chunks are transmitted. The atomicity invariant is that all chunks are sent from the `File Processor` to the `Listener` or none are.

6.1 Evaluation

To measure the cost of injecting speculations into `Swerve`, we hosted and requested the `javaDocs` API in HTML format. We manually injected timeouts every ten requests and made sure no other request is timed-out by setting the timeout quantum large enough for the remaining nine requests. Our measurements are an average of one hundred runs. The average file sized per request was about 10K.

The results are shown in Table 1; the most notable are total overheads for runtime and memory. The cost of monitoring transactions and their transitive dependencies adds on average about three percent runtime overhead and about five percent memory overhead. On average about nine transactions are affected when a timeout occurs, a regular request usually utilizes about fifteen transactions. Of the 247 channels `Swerve` uses to communicate between threads, on average 4 must be cleared of non-isolated values when the request is aborted. Similarly, about four shared memory locations must be reverted per revoked transaction and 12 memory location induce dependencies between transactions.

7 Related Work

Recent work on safe futures for Java [22] bear some similarity to the non-isolated transactions defined here insofar as both provide a revocation mechanism based on tracking dynamic data and control-flow. However, safe futures do not compose with other Java concurrency primitives, and the criteria for revocation is automatically determined based on dependency violations, and is thus not under user control.

Thread level speculation (TLS) techniques provide safety guarantees on the execution of speculative threads [5, 19]. TLS is typically implemented using one of two main strategies: helper threads and explicitly parallel speculative threads. These threads are injected by the compiler and often require special hardware, such as a speculative cache coherency protocols or explicit write buffers, to provide safety guarantees. Helper threads [2, 16] are used to help reduce the cost of high latency instructions but do not modify processor state. Such threads are mainly used to precompute load addresses or branch directions, but all modifiable state is computed by a non-speculative thread.

Our work follows a long line of previous research investigating higher-level concurrency abstractions for programming next-generation parallel architectures [8, 9, 3, 12, 6, 24, 7, 4], However, since we provide support for relaxed isolation, our design permits a greater degree of interaction among transactions than currently possible. Selective breakage of isolation can often be useful both to exploit available processor cycles, and to support programming idioms that require communication among concurrently executing threads (e.g., producer/consumer relationships). Transaction aborts still occur when serializability violations are detected; however, when such violations do occur, the implementation automatically tracks and reverts any other transaction that may have witnessed data produced by the aborted one. Although transactions offer the ability to bypass isolation through open nesting [13, 20, 14, 11, 14], users must specify compensations to revert any state which escapes the nested transaction. Harris *et al.* proposes a transactional memory system [7] for Haskell that introduces a `retry` primitive to allow a transactional execution to safely abort and be re-executed if desired resources are unavailable. However, this work does not propose to track or revert effectful thread interactions within a transaction.

References

[1] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. Simultaneous subordinate microthreading (SSMT). In *ISCA*, pages 186–195, 1999.

[2] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher J. Hughes, Yong fong Lee, Dan Lavery, and John P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th Annual International Symposium On Computer Architecture*, pages 14–25, 2001.

[3] Jim Gray and Andreas Reuter. *Transaction Processing*. Morgan-Kaufmann, 1993.

[4] Nicolas Haines, Darrell Kindred, Gregory Morrisett, Scott Nettles, and Jeannette Wing. Composing First-Class Transactions. *ACM Transactions on Programming Languages and Systems*, 16(6):1719–1736, 1994.

	Channels		Threads		Shared		Graph	Overheads (%)	
	Num	Cleared	Total	Affected	Writes	Reads	Size(MB)	Runtime	Memory
`Swerve`	247	6	10397	9	4	12	5.34	3.28	5.03

Table 1. Instrumented recovery overheads on 100 runs for a concurrent web server.

[5] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 58–69, New York, NY, USA, 1998. ACM Press.

[6] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, 2003.

[7] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable Memory Transactions. In *ACM Conference on Principles and Practice of Parallel Programming*, 2005.

[8] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. 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.

[9] Maurice Herlihy and J. Eliott B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 1993 International Symposium on Computer Architecture*, 1993.

[10] http://www.mlton.org.

[11] Michelle Moravan, Jayaram Bobba, Kevin Moore, Luke Y, Mark Hill, Ben Liblit, Michael Swift, and David Wood. Supporting Nested Transactional Memory in LogTM. In *Architectural Support for Programming Languages and Systems*, 2006.

[12] E. B Moss. Nested transactions: An approach to reliable distributed computing. In *Tech. rep., Massachusetts Institute of Technology*, 1981.

[13] E. B Moss. Open nested transactions: Semantics and support. In *4th Workshop on Memory Performance Issues*, 2006.

[14] Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony Hosking, Richard Hudson, J. Eliot Moss, Bratin Saha, and Tatiana Shpeisman. Open Nesting in Software Transactional Memory. In *ACM Symposium on Principles and Practice of Parallel Programming*, 2007.

[15] John H. Reppy. CML: A Higher-Order Concurrent Language. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 293–305, New York, NY, USA, 1991. ACM Press.

[16] Amir Roth and Gurindar S. Sohi. Speculative data-driven multithreading. In *HPCA '01: Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, page 37, Washington, DC, USA, 2001. IEEE Computer Society.

[17] Nir Shavit and Dan Touitou. Software Transactional Memory. In *ACM Conference on Principles of Distributed Computing*, 1995.

[18] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium On Computer Architecture*, pages 414–425, New York, NY, USA, 1995. ACM Press.

[19] J. Steffan and T Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *HPCA '98: Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 2, Washington, DC, USA, 1998. IEEE Computer Society.

[20] Gerhard Weikum and Hans-Jorg Schek. Concepts and applications of multilevel transactions and open nested transactions. In *Database Transaction Models for Advanced Applications*, pages 515–553. Morgan-Kaufmann, 1992.

[21] Adam Welc, Antony Hosking, and Suresh Jagannathan. Transparently Reconciling Transactions with Locking for Java Synchronization. In *European Conference on Object-Oriented Programming*, pages 148–173, 2006. LNCS 4067.

[22] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 439–453. ACM Press, 2005.

[23] Adam Welc, Suresh Jagannathan, and Antony Hosking. Revocation Techniques for Java Concurrency. *Concurrency and Computation: Practice and Experience*, January 2006. published online.

[24] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional monitors for concurrent objects. In Martin Odersky, editor, *Proceedings of the European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 519–542. Springer-Verlag, 2004.

[25] Lukasz Ziarek, Philip Schatz, and Suresh Jagannathan. Stabilizers: A Modular Checkpointing Abstraction for Concurrent Functional Programs. In *ACM International Conference on Functional Programming*, 2006.