Towards Transactional Memory Support for GCC

Martin Schindewolf, Albert Cohen, Wolfgang Karl, Andrea Marongiu, Luca Benini

Towards Transactional Memory Support for GCC

Martin Schindewolf[1], Albert Cohen[2], Wolfgang Karl[1], Andrea Marongiu[3], and Luca Benini[3]

[1] Institut für Technische Informatik
Zirkel 2
76131 Karlsruhe, Germany
[2] INRIA Saclay – Île-de-France
Parc Orsay Université
4 rue Jacques Monod
91893 Orsay Cedex, France
[3] DEIS – University of Bologna
Via Risorgimento 2
40136 Bologna, Italy
{schindew,karl}@ira.uka.de
{Albert.Cohen}@inria.fr
{a.marongiu,luca.benini}@unibo.it

Abstract. Transactional memory is a parallel programming model providing many advantages over lock-based concurrency. It is one important attempt to exploit the potential of multicore architectures while preserving software development productivity. This paper describes the design of a transactional memory extension for GCC, and highlights research challenges and perspectives enabled by this design.

Key words: GCC, Transactional Memory, STM

1 Introduction

Despite massive investments in Transactional Memory (TM) research and development, the academic and industrial proposals have not yet converged towards a broadly accepted language semantics. This proves the vitality and originality of the ongoing research, but it delays the emergence of production-quality, TM-enabled compilers and TM-based parallel applications. This is a major roadblock for wider adoption of TM mechanisms by the software industry. This in turn restricts the relevance of the few available benchmarks, impacting the methodological soundness of the majority of TM work.

This paper describes the design of a transactional extension for the C language, implemented in the GNU Compiler Collection (GCC). This design derives from the pioneering work of Intel [1]. This work has recently led to an important standardization effort let by Ali-Reza Adl-Tabatabai, from the syntax to the Application Binary Interface (ABI), through the semantics (and memory model) and interactions with existing programming languages and practices.

Participating to this important standardization effort is a necessary step towards a mature TM technology, upon which software developers and parallel computing research depend. In this context, we highlight some important ongoing research opportunities and challenges.

Transactional memory is a set of a parallel programming construct and the accompanying programming patterns [6, 5]. It borrows database semantics, terminology and designs to address the atomic execution problem. In contrast to traditional low-level synchronization mechanisms, the programmer does not manage locks directly but relies on a more abstract, structured concept: an *atomic block*, hereafter called a *transaction*. From the programmer's point of view, atomicity is understood as two-way isolation of shared memory loads and stores within a transaction. From an implementation point of view, it allows for *optimistic concurrency*, with speculative conversion of the coarse-grain critical section into finer-grain, object-specific ones. The ability to correctly and efficiently transpose coarse-grain transactions into fine-grain, speculative concurrency is the key challenge for TM research and development. Both semantical and performance issues lead to a vast amount of studies and results [9]. Because of this implicit support for speculative execution, TM programming patterns generally include *failure atomicity* mechanisms, with programmer-controlled *abort* and *retry* constructs. These constructs are, for a part, complementary to parallel programming, and can improve the software development productivity at large.

Based on this design and implementation effort, we are conducting research on compiler optimizations to reduce the performance penalty of STM systems. We also study the potential of TM to support automatic parallelization, enhancing the support for generalized and sparse reductions in the automatic parallelization pass of GCC.

The structure of the paper is the following. Section 2 discusses related work in the area of compiler support for transactional memory. Section 3 presents the design and implementation in GCC; it also reviews ongoing research and development regarding TM-aware and TM-specific compiler optimizations. Section 4 discusses more long term research opportunities, before we come to some preliminary conclusions in Section 5.

2 Related Work

Let us discuss the most closely related work, starting with the papers that influenced our semantical choices and compilation strategy. This paper studies TM in the context of unmanaged languages only, with a *word-based instrumentation of shared memory accesses* in transactions. In this context, it is also natural to assume *weak isolation* of transactions with respect to non-transactional code; this comes with obvious limitations in terms of concurrency guarantees and cooperation with legacy code [9].

Many semantical variants of transactions have been proposed and investigated. The baseline semantics in our design is the one of a critical section guarded by a single lock, shared by all transactions. This choice is consistent with the

original concept [6] and with most industrial designs; it offers composability and liveness guarantees, and is the only one for which a sound, intuitive and reasonably efficient weakly-consistent memory model has been proposed [10]. Our design is compatible multiple transactional memory runtimes, facilitating its adoption in research environments and leveraging existing software support.

At compile time a TM-enabled compiler substitutes accesses to shared memory inside transactions with calls to a Software Transactional Memory library (STM). This library may come with hardware support, like in the Sun Rock processor; this design is called Hybrid Transactional Memory (HTM). Those approaches differ significantly in terms of shared memory accesses overhead. In the STM approach, the role of compiler optimizations is paramount to mitigate this overhead [16].

Some researchers propose transactional memory as an enhancement to OpenMP [11, 2]. These proposals include a variety of new transactional directives, such as `#pragma omp sections transaction` grouping together independent sections that are treated as transactions. OpenTM [2] is implemented in GCC and supports two nesting variants: *open* and *closed* nesting. Open nesting publishes the state of an inner transaction in case the outer transaction aborts whereas closed nesting discards the changes from the inner transactions causing no side effects; open nesting allows for additional optimizations to happen at compilation and runtime, but breaks major assumption about transactional execution (it is intended for expert library developers). An extension to the `omp for` directive is also proposed, `omp transfor`, executing the loop iterations in parallel as transactions. Furthermore, the programmer may specify the scheduling of these loop iterations and enforce sequential commit of the transactions (relying on the quiescence mechanism) to enable a memory consistency behavior compatible with weakly isolated, single-lock execution [10]. Milovanović et al. [11] study the interaction between OpenMP 3.0 tasks and transactional execution. In particular, an optional list holds the shared memory locations to instrument or not instrument. This mechanism provides the programmer with a verbose yet effective means to reduce instrumentation overhead. A similar mechanism is proposed in IBM's TM-enabled XL Compiler [8]. We decided not to bind our TM extensions and compiler support to OpenMP, keeping our design as generic and simple as possible. This choice does not contradict future TM extensions of GCC's OpenMP passes and runtime.

Intel develops McRT, a runtime system for multicore architectures, which includes an STM library implementation. It comes with language and compiler support for transactions [16], and transactional versions of C library functions such as `malloc` and `free` [7]. Concluding from their experience with transactional workloads, the overhead of strong isolation [13], and the desired TM properties for the most important concurrent programming patterns, they advocate for a combination of single-lock semantics, weakly isolated transactions and weakly consistent model [10]. This combination also drives our own design as it avoids many performance pitfalls, semantical flaws and unrealistic assumptions on the compilers.

Tanger is an open source compiler framework that supports the use of transactions [4]. It is based on the LLVM (low level virtual machine) intermediate representation and generates code for the TinySTM library [15]. Further enhancements to Tanger allow the conflict detection algorithm of TinySTM to operate on objects in an unmanaged environment [12]. This project influenced our implementation and led to the selection of TinySTM as the first runtime for the TM-enabled GCC.

3 Design

This section presents the design decisions and additional mechanisms for TM support in GCC. One of the major design goals is to be orthogonal to other parallel programming models. Thus, the implementation is not based on OpenMP.

We wish to support the optimistic execution of transactions, in the form of the simple example in Figure 1. To make this possible in C and in GCC, several enhancements are necessary. Besides some minor modifications to the C front-end to add support for the `#pragma tm atomic` and `__tm_abort`, we implemented two compilation passes: the *expansion* and the *checkpointing* pass. New GIMPLE tree codes `GTM_TXN`, `GTM_TXN_BODY`, and `GTM_RETURN` are introduced while parsing the transactified source code. The construction of the control flow graph is altered according to the OpenMP scheme for atomic sections: a basic block is split everytime a `GTM_DIRECTIVE` is encountered; this scheme simplifies the identification and management of transactions during the expansion pass.

```
int gvar;
int main () {
   int a = 15;
#pragma tm atomic
{
   gvar = ++a;
}
   printf ("Global␣variable␣%d\n", gvar);
}
```

Fig. 1. Simple example

3.1 Expansion

The first pass is called `gtm_exp`. It performs the following expansion tasks:

- function cloning, for all functions marked as callable from a transaction;
- recombination of the previously split basic blocks;
- instrumentation of shared-memory loads and stores with calls to the STM runtime — read or write *barriers*.

We currently instrument all pointer-based accesses. GCC's *escape* information will be used to later restrict this instrumentation to shared locations only.

In addition, the pass checks for language restrictions that apply for transactions. For instance, invocations of `__tm_abort` are only valid in the scope of a transaction. To access and process transactions conveniently, a `gtm_region` tree is built. The region tree facilitates the flattening of inner transactions.

3.2 Checkpointing

In case a transaction is rolled back, the effects on registers and stack variables have to be undone. The procedure to revert to the architectural state before entering the transaction consists of a call to `setjmp` combined with saving the contents of variables. We refer to this mechanism as *checkpointing*. An alternative to checkpointing variables, is to copy and restore the active stack frame as described in [4]. When the transaction rolls back the old stack frame is substituted for the new one to restore the previous state. Which of the two approaches is superior depends on the use case. If many variables are live-in to the transaction, copying a continuous amount of memory is expected to be faster than copying each variable exclusively. In case the amount of live-in variables is small compared to the active stack frame, copying and replacing variables is faster. We believe that the latter use case is more common. Thus, the second compiler pass implements the checkpointing scheme similar to the one in [16]. In addition the `setjmp`/`longjmp` mechanism is used to restore the actual register file. During the compiler pass one additional basic block is introduced. This basic block is connected via the control flow so that it is executed in case of a rollback and restores the values of variables. The saving of the values (and storing them into a temporary variable) is done before calling setjmp. In order to reduce the number of copy and restore operations, only variables that are live-in to the transaction are considered. The availability of liveness information require the pass to operate on SSA-form. For a seamless integration with the previous `gtm_exp` pass, the `gtm_checkpoint` pass removes the marker and adds the real checkpointing scheme. The outcome of this procedure is illustrated in Figure 2: the instruction sequence before the call to setjmp captures the value of the live-in variable `a` and saves it into the temporary variable `txn_save_a`. In case the transaction has to roll back, the library executes a call to `longjmp` and returns to the location where the `setjmp` was called. Thus, it returns from the setjmp with a non-zero return value. Subsequently, the basic block on the right hand side of Figure 2 gets executed and the value of the variable is restored to `a`. The Φ node on the next basic block merges the different versions of `a`.

3.3 Optimizations

This paragraph outlines four opportunities and directions for optimization.

First exploiting the properties of the underlying intermediate representation gives some benefits in terms of readability of the source code and introduced temporary variables. For instance, GIMPLE, the intermediate representation

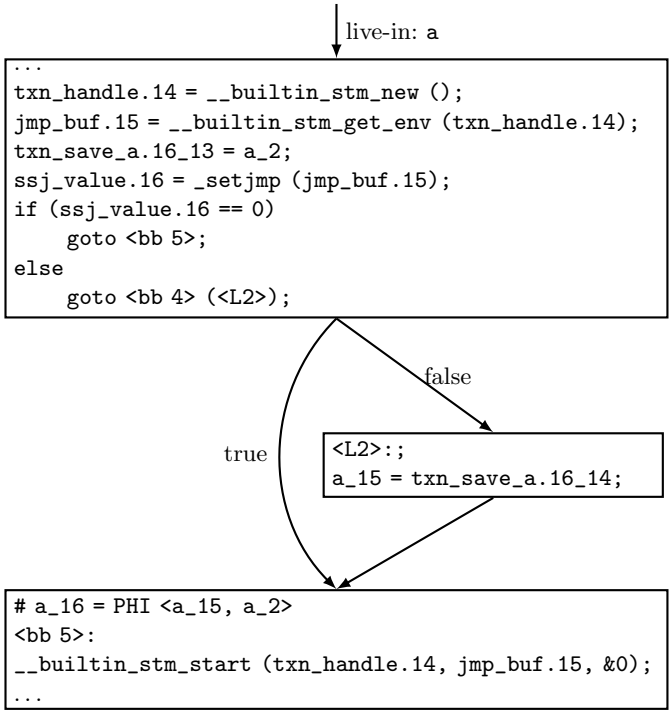

Fig. 2. Checkpointing mechanism after the `gtm_checkpoint` pass.

the `gtm` passes operate on, distinguishes between memory and register loads and stores. Thus, a variable living in memory needs to be loaded into a register prior to being used. Thus, all memory loads are already assigned to a temporary variable on GIMPLE. In order to reduce the number of introduced temporaries, the existing loads and stores could be directly substituted by calls to the STM runtime, reducing the number of temporary variables and, so, the work of optimizers that eliminate temporaries.

Second the STM barriers, represented as builtins (or intrinsics), should make use of the attributes provided within GCC. For all builtin attributes are set in order to allow optimizers to determine the amount of valid optimizations. The current approach is to set attribute `ATTR_NO_THROW_LIST` for all barriers. Relaxing this conservative choice for `stm_load` barriers to `ATTR_PURE_NO_THROW_LIST` seems promising to enable few optimizations while preserving the correctness of the optimized code. Not all STM barriers qualify for relaxed attributes. For instance the `stm_start` and `stm_commit`-barriers enclosing the body of a transaction, are to remain as strict as possible. Otherwise, optimizers, such as the store sinking or load hoisting optimizers may sink stores out of transactions and loads into them, thus. Both optimizations potentially violate the intention of the

programmer and weaken the boundaries set by transactions. Thus, the resulting code would not be correct.

The third and last optimization is to subdivide the passes in order to exploit the optimizations on SSA form. The *expansion* pass would be split into two phases. The remaining first part would only expand the `stm_start` and `stm_commit` barriers. Whereas the second part is placed at the end of the SSA optimization passes and introduces the `stm_load` and `stm_store` barriers. The proposed design utilizes the optimizations on SSA form and respects the properties of transactions.

When transactions occur in OpenMP parallel sections, we may rely on the shared/private clauses to refine the set of variables and locations to be instrumented by memory barriers. This optimization was proposed in previous transactional extensions to OpenMP [11, 2], but it may of course be designed as a best-effort enhancement of our language-independent TM design.

Further design and implementation of these optimization is under way in the context of the `transactional-memory` branch of GCC. This branch initiated from our design, and was opened in October 2008 by Richard Henderson (Red Hat). It uses the same ABI as Intel,[4], its expansion is implemented as an Inter-Procedural Analysis (IPA) pass, and it relies on the Error Handling of GCC for streamlined checkpointing support and conservative interaction with SSA-based optimizations. It is not yet fully functional, but should subsume our initial implementation by the end of 2008.

4 Research Perspectives

This section gives some ideas how potential research is enabled by the support for transactional memory in GCC. The optimistic concurrency exhibited by transactions combined with their guaranteed consistent execution can be exploited to the benefit of many research projects.

Further research may target the optimization of transactional barriers as well as emerging combinations of compiler and run-time support in order to speed up execution time of transactions and investigate trade-offs between compiler and runtime support to implement transactional features. In addition, the current implementation provides the entry point for research concerning the implications from the memory model on transactions with GCC.

The next section shows results demonstrating the potential of combining the automatic parallelization (`parloops`) pass and the TM infrastructure in GCC. The results show the speedup using transactions compared to synchronization primitives based on (POSIX thread) locks and higher level OpenMP critical sections.

4.1 Optimizations and Extensions

Transactional environments require special mechanisms to enable developers to apply common programming patterns. It is the case of the *publication* and *priva-*

[4] `http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition-20#ABI`

tization patterns that frequently arise while programming with locks [10]: they feature concurrent accesses to shared variables inside *and outside* transactions. The absence of races is guaranteed by the lock semantics and by any weak memory model that subsumes Release Consistency (RC).

Semantical support for these patterns is particularly helpful when transactifying legacy code with non-speculative critical sections. Indeed, weak isolation and weak memory consistency models do not guarantee that such publication and privatization patterns will behave consistently with a lock-based implementation. Current STM designs propose *quiescence* as the mechanism to solve the problem occuring while one transaction tries to privatize a data member whereas the other tries to write into it [10]. Quiescence enforces an ordering of transactions so that transactions complete in the same order as they started. Besides allowing the programmer to use well known constructs and follow classical programming patterns, this mechanism comes with a significant performance penalty. We believe that further research in this area is inevitable to speed up the execution of transactions while retaining a consistency model compatible with easy transactification of lock-based code — here *single-lock semantics* is sufficient [10].

Calling legacy code from inside transactions constitutes another problem for programming in a transactional environment, because effects of these functions can not be rolled back. The same holds true for system calls. The solution is to let transactions execute in, or transition to, *irrevocable mode*. The runtime assures that the irrevocable transaction is the only one executing and, thus, can not conflict with other transactions. Hence, the transaction runs to completion. [17] presents possible implementations and applications of irrevocable transactions, whereas [14] also evaluates different optimized strategies to implement irrevocability. Further research concerning irrevocability could benefit from the presented implementation.

Link-Time Optimization (LTO) as well as Just-In-Time (JIT) compilation are well-known compilation approaches that are not yet extensively applied to transactional workloads. The former has a high potential for pointer-analysis-based optimizations (like escape analyses to eliminate unneccessary barriers), while the latter can substiture dynamic code generation and transaction instrumentation rather than static cloning of functions callable from transactions.

4.2 Parallelization of Irregular Reductions

Reduction operations are a computational structure frequently found in the core of many irregular numerical applications. A reduction is defined from associative and commutative operators acting on simple variables (scalar reductions) or array elements inside a loop (histogram reductions). If there are no other dependencies but those caused by reductions, the loop can be transformed to be executed fully parallel, since — due to the associativity and commutativity of their operands — iterations of a reduction loop can be reordered without affecting the correctness of the final result.

Currently, the automatic loop parallelization pass in GCC is capable of recognizing scalar reductions. Once the reduction pattern has been detected the

code generation step relies on the OpenMP expansion machinery to distribute iterations of the loop into several threads. Reduction parallelization employs a privatization algorithm: the transformed loop has a parallel prefix, where each thread accumulates partial results in local copies of the reduction variable, followed by a cross-thread merging phase in which partial results are combined into the shared (reduction) variable.

The reduction recognition routine in the automatic parallelization pass can be extended to detect *sparse* reductions. Sparse reductions correspond to inductive dependences on the reduction variable/array that only exists for a fraction of the loop iterations. This is generally the case for moderate-to-large reductions with indirection variables (e.g., histograms), as well as some reductions guarded with data-dependent control flow.

A parallel reduction algorithm has to be chosen for the code generation of sparse reductions. A simple solution is that of enclosing the accesses to the reduction variable/array within a critical section. The main drawback of the typical lock-based implementation of this method is that it exhibits a very high synchronization overhead. We can leverage the infrastructure for transactional memory programming within GCC to devise an alternative approach to reduction parallelization in which we enclose the critical section in an atomic transaction, and let the underlying STM runtime detect and resolve possible conflicting accesses to same array locations. Many scientific and numerical applications operate on large and sparse datasets; they are amenable to transactional parallelization since we can optimistically assume that only few conflicting accesses to the same memory locations will manifest at runtime.

As a motivating example we show here the results of parallelization of a synthetic loop containing a histogram reduction, see Figure 3. To model the effect of varying amount of work besides the reduction within the loop, we employ a WORK section consisting in an additional loop nest which iterations have been parameterized with variable M. This loop only operates on data independent on the reduction operation. Loop iterations are distributed between 4 threads, and we compare three different synchronization schemes, namely locks (pthreads), OpenMP `critical` directive and transactions. The latter is achieved through the use of the `#pragma tm atomic`. Calls to the STM library (TinySTM v.0.9.0b1 [15]) for read/write barrier instrumentation and transaction rollback/restart are automatically instantiated by the GTM compiler.

To account for the effect of different degrees of contention we consider histogram creation for two synthetic images: a completely black image (our worst case), and an image with randomly generated pixel values.

We show in Figure 4 the results of the execution of the example loop on a Intel Core 2 Quad CPU (4MB L2 cache). On the X-axis we consider increasing amounts of work in the loop body by increasing the value of the parameter **M**. On the Y-axis we plot the speedup of the various parallelization schemes against the serial version of the loop.

In the random image there is low contention for array locations, and the performance of the optimistic TM approach is always better than the others. We

```
int image[1024][768];
int main ()
{
  int hist[256];

#pragma omp parallel for
  for (i=0, i<1024; i++)
    for (j=0; j<768; j++)
    {
      /* Some reduction-independent
       * work parameterized with M
       */
      WORK;
      pixval = image[i][j];
      /* Begin critical section */
      hist[pixval]++;
      /* End critical section */
    }
}
```

Fig. 3. Example of reduction

can achieve speedups with this technique even for small values of M. However, as expected, high contention on array elements has a strong impact on the performance of TM. This can be seen comparing the trend of the TM curve in the two plots, and is justified by the fact that the overhead for frequent transaction rollback and restart is greater than that carried by the locks. Clearly this behavior is also affected by the value of M. When there is little work besides the reduction in the loop the overhead is predominant for all of the proposed techniques, but in high-contention scenarios TM is the one that is mostly affected by this parameter, not only because it influences the frequency of aborts and rollbacks, but also because the overhead for starting and committing a transaction is not amortized by other computation.

These two factors are directly related to the sparsity of the dataset on which we operate and to the granularity of the transaction, for this reason we consider them as the two main parameters to be investigated in real workloads to discover the boundaries wherein a reduction parallelization algorithm that exploits transactions is successful. First experimental results are encouraging, since they show that adequately tuning these parameters transactional approach to irregular reduction parallelization can bring significant performance improvements with respect to the use of locks.

5 Conclusion

We presented a transactional memory extension of the GNU Compiler Collection, and stressed its language-independent and STM-oriented design (yet compatible with hybrid hardware/software implementations). We also highlighted key optimization challenges and opportunities; together with Yoo et al. [18] and the more pessimistic study of Cascaval et al. [3], we stress the importance of

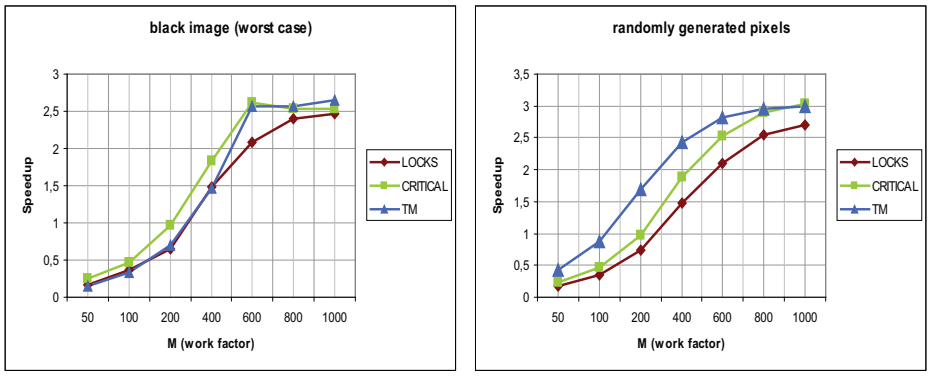

Fig. 4. Performance comparison of transactional and lock-based parallel reductions

compiler and joint language-compiler studies for the future adoption of TM in real world applications. We believe that GCC's infrastructure is well suited to address the 4 main issues identified by Yoo et al. [18]:

1. false conflicts;
2. over instrumentation;
3. quiescence;
4. overhead amortization.

We also initiated research towards integrating TM in an enhanced automatic parallelization strategy, where much of its design and implementation can be reused for the parallelization of sparse, generalized reductions. In this context, and together with failure atomicity (explicit abort), TM may also be used as a runtime support for speculative execution.

References

1. Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI'06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM.
2. Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun. The OpenTM transactional application programming interface. In *PACT'07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 376–387, Washington, DC, USA, 2007. IEEE Computer Society.
3. Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: why is it only a research toy? *Commun. ACM*, 51(11):40–46, 2008.

4. Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin Süßkraut, and Heiko Sturzrehm. Transactifying applications using an open compiler framework. In *TRANSACT*, August 2007.
5. Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *PPOPP*, pages 48–60, 2005.
6. Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.
7. Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. McRT-Malloc: a scalable transactional memory allocator. In *ISMM'06: Proceedings of the 5th international symposium on Memory management*, pages 74–83, New York, NY, USA, 2006. ACM.
8. IBM. IBM XL C/C++ for transactional memory for AIX. `http://www.alphaworks.ibm.com/tech/xlcstm`, June 2008.
9. James R. Larus and Ravi Rajwar. *Transactional memory*. Morgan & Claypool, 2007.
10. Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard Hudson, Bratin Saha, and Adam Welc. Practical weak-atomicity semantics for Java STM. In *SPAA'08: 20th ACM Symposium on Parallelism in Algorithms and Architectures*, June 2008.
11. Miloš Milovanović, Roger Ferrer, Vladimir Gajinov, Osman S. Unsal, Adrian Cristal, Eduard Ayguadé, and Mateo Valero. Multithreaded software transactional memory and OpenMP. In *MEDEA'07: Proceedings of the 2007 workshop on MEmory performance*, pages 81–88, New York, NY, USA, 2007. ACM.
12. Torvald Riegel and Diogo Becker de Brum. Making object-based stm practical in unmanaged environments. In *TRANSACT 2008*, 2008.
13. Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. *SIGPLAN Not.*, 42(6):78–88, 2007.
14. Michael F. Spear, Michael Silverman, Luke Dalessandro, Maged M. Michael, and Michael L. Scott. Implementing and exploiting inevitability in software transactional memory. *Parallel Processing, International Conference on*, 0:59–66, 2008.
15. TinySTM. `http://tinystm.org/tinystm`. Online, last modified 30th of May 2008.
16. Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO'07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 34–48, Washington, DC, USA, 2007. IEEE Computer Society.
17. Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In *SPAA'08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 285–296, New York, NY, USA, 2008. ACM.
18. Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *SPAA'08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 265–274, New York, NY, USA, 2008. ACM.