The OpenTM Transactional Application Programming Interface

Woongki Baek, Chi Cao Minh, Martin Trautmann,
Christos Kozyrakis and Kunle Olukotun
Computer Systems Laboratory
Stanford University
{*wkbaek, caominh, mat42, kozyraki, kunle*}*@stanford.edu*

Abstract

Transactional Memory (TM) simplifies parallel programming by supporting atomic and isolated execution of user-identified tasks. To date, TM programming has required the use of libraries that make it difficult to achieve scalable performance with code that is easy to develop and maintain. For TM programming to become practical, it is important to integrate TM into familiar, high-level environments for parallel programming.

This paper presents OpenTM, an application programming interface (API) for parallel programming with transactions. OpenTM extends OpenMP, a widely used API for shared-memory parallel programming, with a set of compiler directives to express non-blocking synchronization and speculative parallelization based on memory transactions. We also present a portable OpenTM implementation that produces code for hardware, software, and hybrid TM systems. The implementation builds upon the OpenMP support in the GCC compiler and includes a runtime for the C programming language.

We evaluate the performance and programmability features of OpenTM. We show that it delivers the performance of fine-grain locks at the programming simplicity of coarse-grain locks. Compared to transactional programming with lower-level interfaces, it removes the burden of manual annotations for accesses to shared variables and enables easy changes of the scheduling and contention management policies. Overall, OpenTM provides a practical and efficient TM programming environment within the familiar scope of OpenMP.

1 Introduction

Transactional Memory (TM) [16] is emerging as a promising technology to address the difficulty of parallel programming for multi-core chips. With TM, programmers simply declare that certain code sections should execute as *atomic and isolated transactions* with respect to all other code. Concurrency control as multiple transactions execute in parallel is the responsibility of the system. Several TM systems have been proposed, based on hardware [21, 24], software [26, 11], or hybrid techniques [18, 27, 6].

To achieve widespread use, TM must be integrated into practical and familiar programming environments. To date, TM programming has primarily been based on libraries that include special functions to define transaction boundaries, manipulate shared data, and control the runtime system. While the library-based approach is sufficient for initial experimentation with small programs, it is inadequate for large software projects by non-expert developers. Library-based code is difficult to comprehend, maintain, port, and scale, since the programmer must manually annotate accesses to shared data. Library annotations can also complicate static analysis and reduce the effectiveness of compiler optimizations [2].

This paper presents *OpenTM*, an application programming interface (API) for parallel programming with transactions. OpenTM extends the popular OpenMP API for shared-memory systems [1] with the compiler directives necessary to express both *non-blocking synchronization* and *speculative parallelization* using memory transactions. OpenTM provides a simple, high-level interface to express transactional parallelism, identify the role of key variables, and provide scheduling and contention management hints. OpenTM code can be efficiently mapped to a variety of hardware (HTM), software (STM), and hybrid TM implementations. It can also be tuned by an optimizing compiler and dynamically scheduled by the runtime system.

The specific contributions of this work are:

- We define the set of extensions to OpenMP that support both non-blocking synchronization and speculative parallelization using transactional memory techniques. Apart from integrating memory transactions with OpenMP constructs such as parallel loops and sections, the extensions address issues such as nested transactions, conditional synchronization, scheduling, and contention management.

- We describe an OpenTM implementation for the C programming language. The environment is based on the OpenMP support in the GCC compiler [25] and produces executable code for hardware, software, and hybrid TM systems. The implementation is easy to port to any TM system that provides a basic low-level interface for transactional functionality.

- We evaluate the performance and programmability features of parallel programming with OpenTM. Using a variety of programs, we show that OpenTM code is simple, compact, and scales well.

The rest of the paper is organized as follows. Section 2 reviews the OpenMP API, while Section 3 introduces the OpenTM extensions. Section 4 describes our first OpenTM implementation. Sections 5 and 6 present the quantitative evaluation. Section 7 reviews related work, and Section 8 concludes the paper.

2 OpenMP Overview

OpenMP is a widely used API for shared-memory parallel programming [1]. The specification includes a set of compiler directives, runtime library routines, and environment variables. OpenMP follows the *fork-join* parallel execution model. The *master thread* begins a program execution sequentially. When the master thread encounters a **`parallel`** construct, it creates a set of *worker threads*. All workers execute the same code inside the **`parallel`** construct. When a work-sharing construct is encountered, the work is divided among the concurrent workers and executed cooperatively. After the end of a work-sharing construct, every thread in the team resumes execution of the code in the **`parallel`** construct until the next work-sharing opportunity. There is an implicit barrier at the end of the **`parallel`** construct, and only the master thread executes user code beyond that point.

The OpenMP memory model is shared memory with relaxed consistency [3]. All threads can perform load and store accesses to variables in shared memory. The **`parallel`** directive supports two types of variables within the corresponding structured block: *shared* and *private*. Each variable referenced within the block has an original variable with the same name that exists outside of the **`parallel`** construct. Accesses to shared variables from any thread will access the original variable. For private variables, each thread will have its own private copy of the original variable.

OpenMP provides programmers with five classes of directives and routines: parallel, work-sharing, synchronization, data environment, and runtime library routines. The **`parallel`** directive starts a parallel construct. The work-sharing directives can describe parallel work for iterative (loops) and non-iterative (parallel sections) code patterns. The following is a simple example of a parallel loop (forall) in OpenMP:

```
#pragma omp parallel for
  for(i=0;i<n;i++)
    b[i]=(a[i]*a[i])/2.0;
```

Synchronization constructs and routines, such as **`critical`** and **`barrier`**, allow threads to coordinate on shared-memory accesses. Data environment primitives describe the sharing attributes of variables referenced in parallel regions. Finally, runtime routines specify a set of interfaces and variables used for runtime optimizations, synchronization, and scheduling. A detailed specification and several tutorials for OpenMP are available in [1].

3 The OpenTM API

OpenTM is designed as an extension of OpenMP to support non-blocking synchronization and speculative parallelization using transactional techniques. Hence, OpenTM inherits the OpenMP execution model, memory semantics, language syntax, and runtime constructs. Any OpenMP program is a legitimate OpenTM program. Non-transactional code, parallel or sequential, behaves exactly as described in the OpenMP specification. We discuss the interaction of the new OpenTM features with certain OpenMP features in Section 3.5.

3.1 OpenTM Transactional Model

The transactional model for OpenTM is based on three key concepts.

Implicit Transactions: OpenTM supports implicit transactions. The programmer simply defines transaction boundaries within parallel blocks without any additional annotations for accesses to shared data. All memory operations are executed implicitly on transactional state in order to guarantee atomicity and isolation. Implicit transactions minimize the burden on programmers but require compiler and/or hardware support in order to implement transactional bookkeeping. For STM systems, the compiler inserts the necessary barriers, while for HTM systems, the hardware performs bookkeeping in the background as the transaction executes regular loads and stores [2]. Even with hardware support, compiler optimizations can be highly profitable. Another advantage of implicit transactions is that they simplify software reuse and composability. A programmer can reuse a function with implicit transactions without having to reconsider the barriers necessary for transactional bookkeeping under the new context.

Strong Isolation: In OpenTM programs, memory transactions are atomic and isolated with respect to other transactions and non-transactional accesses. There also exists a consistent ordering between committed transactions and non-transactional accesses for the whole system. This property is known as *strong isolation* and is necessary for correct

and predictable interactions between transactional and non-transactional code [19]. All hardware [21, 24] and some hybrid [6] TM systems guarantee strong isolation. For software TM systems, the compiler must insert additional read and write barriers outside of transactions in order to implement strong isolation [29].

Virtualized Transactions: OpenTM requires that the underlying TM system supports *virtualized transactions* that are not bounded by execution time, memory footprint, and nesting depth. The TM system should guarantee correct execution even when transactions exceed a scheduling quantum, exceed the capacity of hardware caches or physical memory, or include a large number of nesting levels. While it is expected that the majority of transactions will be short-lived [4, 10, 6], programmers will naturally expect that any long-lived transactions, even if infrequent, will be handled correctly in a transparent manner. Virtualization is a challenge for HTM systems that use hardware caches and physical addresses for transactional bookkeeping. Excluding the performance implications, OpenTM is agnostic to the exact method used to support virtualized transactions [4, 9, 8].

3.2 Basic OpenTM Constructs

The following are the basic OpenTM extensions for parallel programming with transactions.

Transaction Boundaries: OpenTM introduces the **`transaction`** construct to specify the boundaries of strongly isolated transactions. The syntax is:

`#pragma omp transaction` *[clause[[,] clause]...]*
structured-block

where *clause* is one of the following: **`ordered`**, **`nesting(open|closed)`**. **`ordered`** is used to specify a sequential commit order between executions of this transaction by different threads. This is useful for speculative parallelization of sequential code. If not specified, OpenTM will generate unordered yet serializable transactions. Unordered transactions are useful for non-blocking synchronization in parallel code. During the execution of transactions, the underlying TM system detects conflicting accesses to shared variables in order to guarantee atomicity and isolation. On a conflict, the system aborts and re-executes transactions based on the ordering scheme and a contention management policy. We discuss the **`nesting`** clause along with the advanced OpenTM features in Section 3.3. Note that the definition of *structured-block* is the same as in OpenMP.

Transactional Loop: The **`transfor`** construct specifies a loop with iterations executing in parallel as atomic transactions. The **`transfor`** syntax is:

`#pragma omp transfor` *[clause[[,] clause]...]*
for-loop

`transfor` reuses most of the clauses of the OpenMP **`for`** construct such as **`private`** and **`reduction`** to identify private or reduction variables, respectively. Certain clauses are extended or added to specify the transactional characteristics of the associated loop body. The **`ordered`** clause specifies that transactions will commit in sequential order, implying a foreach loop with sequential semantics (speculative loop parallelization). If **`ordered`** is not specified, **`transfor`** will generate unordered transactions, which implies an unordered foreach loop with potential dependencies. The OpenMP **`for`** construct specifies a forall loop.

The **`transfor`** construct can have up to three parameters in the **`schedule`** clause. Just as with the OpenMP **`for`**, the first parameter specifies how loop iterations are scheduled across worker threads (see discussion in Section 3.4). The second parameter identifies the number of iterations (*chunk size*), assigned to each thread on every scheduling decision. The tradeoff for chunk size is between scheduling overhead and load balance across threads. The third parameter specifies how many loop iterations will be executed per transaction by each worker thread (*transaction size*). If it is not specified, each iteration executes as a separate transaction. The tradeoff for transaction size is between the overhead of starting/committing a transaction and the higher probability of conflicts for long-running transactions.

The following code example uses a **`transfor`** to perform parallel histogram updates. In this case, iterations are statically scheduled across threads with a chunk size of 42 iterations. Transactions are unordered with 6 iterations per transaction.

```
void histogram(int *A,int *bin){
 #pragma omp transfor schedule(static,42,6)
 for(i=0;i<NUM_DATA;i++){
  bin[A[i]]++;}}
```

A user can also define transactions using the **`transaction`** construct within the loop body of an OpenMP **`for`** construct. This approach allows programmers to write tuned code with transactions smaller than a loop body that may reduce the pressure on the underlying TM system. On the other hand, it requires better understanding of the dependencies within the loop body and slightly more coding effort.

Transactional Sections: OpenTM supports transactions in parallel sections (non-iterative parallel tasks) using the **`transsections`** construct. Its syntax is:

`#pragma omp transsections` *[clause[[,] clause]...]*
  *[***`#pragma omp transsection`***]*
    structured-block 1
  *[***`#pragma omp transsection`***]*
    structured-block 2
  . . .

Compared to OpenMP **sections**, **transsections** uses an additional **ordered** clause to specify sequential transaction ordering. While **sections** implies that the structured blocks are proven to be parallel and independent, **transsections** can express parallel tasks with potential dependencies. Similar to the loop case, transactional sections can also be specified using the **transaction** construct within OpenMP **sections**.

The following is a simple example of method-level speculative parallelization using OpenTM **transsections** construct:

```
#pragma omp transsections ordered {
  #pragma omp transsection
    WORK_A();
  #pragma omp transsection
    WORK_B();
  #pragma omp transsection
    WORK_C();}
```

3.3 Advanced OpenTM Constructs

The basic OpenTM constructs discussed above are sufficient to express the parallelism in a wide range of applications. Nevertheless, OpenTM also introduces a few advanced constructs to support recently proposed techniques for TM programming. These constructs require advanced features in the underlying TM system.

Alternative execution paths: The **orelse** construct supports alternative execution paths for aborted transactions [15, 2]. The syntax is:

```
#pragma omp transaction
  structured-block 1
#pragma omp orelse
  structured-block 2
...
```

When the transaction for block 1 successfully commits, the entire operation completes and block 2 never executes. If the transaction for block 1 aborts for any reason, the code associated with the **orelse** construct is executed as an atomic transaction. A program can cascade an arbitrary number of **orelse** constructs as alternative execution paths.

Conditional synchronization: OpenTM supports conditional synchronization in atomic transactions using the **omp_retry()** runtime routine. **omp_retry()** indicates that the transaction is blocked due to certain conditions [15]. The runtime system will decide whether the transaction will be re-executed immediately or the corresponding thread will be suspended for a while. The transaction can use the **omp_watch()** routine to notify the runtime system that it should monitor a set of addresses and re-execute the blocked transaction when one of them has been updated [7]. The following is a simple example of the conditional synchronization within a transaction:

```
#pragma omp transaction {
  if (queue.status == EMPTY) {
    omp_watch(addr);
    omp_retry();
  } else {
    t = dequeue(queue);}
```

Compared to conditional/guarded atomic blocks or condition variables [14, 2], conditional synchronization with *retry* allows for complex blocking conditions that can occur anywhere within the transaction. Moreover, the directive-based OpenMP approach places additional restrictions in the specification of the blocking condition. For example, a programmer cannot use array indices to specify the condition. Finally, *retry* is composable [15]. We should note that when **omp_retry()** is called in a transaction that also uses **orelse**, the transaction is aborted and control is transfered immediately to the alternative execution path.

Nested Transactions: The **nesting** clause specifies the behavior of nested transactions. If **nesting** is not specified, OpenTM uses closed-nested transactions by default. Closed-nested transactions can abort due to dependencies without causing their parent to abort [22]. The memory updates of closed-nested transactions become visible to other threads only when the outermost transaction commits. The **open** clause allows a program to start an open-nested transaction that can abort independently from its parent but makes its updates visible immediately upon its commit, regardless of what happens to the parent transaction [22]. Open-nested transactions may require finalizing and compensating actions that execute when the outermost transaction commits or aborts, respectively [13]. While we do not expect that many programmers will use open-nested transactions directly, they can be helpful with addressing performance issues and the implementation of additional programming constructs [22].

Transaction Handlers: The **handler** construct specifies software handlers that are invoked when a transaction commits or aborts. OpenTM handlers follow the semantics presented in [22]. The handler syntax is:

```
#pragma omp transaction [clause[[,] clause]...]
  structured-block 1
#pragma omp handler clause
  structured-block 2
```

where clause is one of the following: **commit**, **abort**, **violation**. Violation refers to a rollback triggered by a dependency, while abort is invoked by the transaction itself. The **handler** construct can be associated with **transaction**, **transfor**, **transsections**, and **orelse** constructs. The code below provides an example with an abort handler used to compensate for an open-nested transaction:

Routine	Description
`omp_in_transaction()`	Return *true* if executed within transaction.
`omp_get_nestinglevel()`	Return the nesting-level of the current transaction.
`omp_abort()`	User-initiated abort of the current transaction.
`omp_retry()`	User-initiated retry of the current transaction.
`omp_watch()`	Add an address to a watch-set [7].
`omp_set_cm()`	Set the contention management scheme.
`omp_get_cm()`	Return the current contention management scheme.

Table 1. The extra runtime routines in OpenTM.

```
#pragma omp transaction {
  #pragma omp transaction nesting(open)
  {
    WORK_A();
  } //end of open-nested transaction
  #pragma omp handler abort
  {
    COMPENSATE_WORK_A();
  }} //end of parent transaction
```

3.4 Runtime System

OpenTM also extends the runtime system of OpenMP to support transactional execution. Table 1 summarizes the additional runtime library routines available to programmers.

Loop Scheduling: The **`for`** construct in OpenMP provides four options for scheduling iterations across worker threads: **`static`**, **`dynamic`**, **`guided`**, and **`runtime`**. Static scheduling statically distributes work to threads, while dynamic scheduling assigns a chunk of iterations to threads upon request during runtime. Guided scheduling is similar to dynamic scheduling, but the chunk size decreases exponentially to avoid work imbalance. Runtime scheduling makes the decision dependent on the *run-sched-var* environment variable at runtime [1]. OpenTM reuses these options but adds an additional task of grouping loop iterations into transactions. When grouping in a dynamic or guided manner, the system can use runtime feedback to minimize the overhead of small transactions without running into the virtualization overhead or the higher probability of conflicts of larger transactions.

Contention Management: OpenTM provides two runtime routines, presented in Table 1, to control the contention management scheme of the underlying TM system [12, 28]. The **`omp_get_cm()`** routine returns the type of the currently used contention management scheme. The **`omp_set_cm()`** routine allows the user to the change contention management scheme for the whole system in runtime. Programmers can use this interface to adapt contention management to improve performance robustness or provide fairness guarantees [12]. The exact parameters for **`omp_set_cm()`** depend on the available policies. For example, our current implementation supports a simple backoff scheme [28] that requires a parameter to specify the maximum number of retries before an aborted transaction gets priority to commit. As TM systems mature, the corresponding contention management techniques will be integrated into OpenTM.

3.5 Open Issues and Discussion

There are certain subtle issues about OpenTM. Our initial specification takes a conservative approach in several cases. However, we expect that practical experience with transactional applications, further developments in TM research, and advanced compiler support, will provide more sophisticated solutions for future versions of OpenTM.

OpenMP Synchronization: OpenTM does not allow the use of OpenMP synchronization constructs within transactions (e.g., **`critical`**, **`atomic`**, **`mutex`**, and **`barrier`**). OpenMP synchronization constructs have blocking semantics and can lead to deadlocks or violations of strong isolation when used within transactions. We also disallow the use of transactions within OpenMP synchronization constructs, as there can be deadlock scenarios if **`omp_retry()`** is used. In general, separating transactional code from blocking synchronization will help programmers reason about the correctness of their code at this point. The blocking semantics of **`atomic`** and **`critical`** are also the primary reason we introduced the non-blocking **`transaction`** construct. Reusing **`atomic`** or **`critical`** for TM programming could lead to deadlocks or incorrect results for existing OpenMP programs.

I/O and System Calls: OpenMP requires that any libraries called within parallel regions are *thread-safe*. Similarly, OpenTM requires that any libraries called within transactions are *transaction-safe*. The challenge for transactional systems is routines with permanent side effects such as I/O and system calls. Currently, OpenTM does not propose any specification for such routines. Each OpenTM implementation is responsible for specifying which library calls can be safely used within transactions as well as what are the commit and abort semantics. There is ongoing research on how to integrate I/O and system calls with transactions using buffering and serialization techniques [22].

Nested Parallelism: For the moment, we do not allow user code to spawn extra worker threads within a transaction.

Relaxed Conflict Detection: Recent papers have proposed directives that exclude certain variables from conflict detection (e.g., **`race`** [31] or **`exclude`** [23]). The motivation is to reduce the TM bookkeeping overhead and any unnecessary conflicts. At the moment, OpenTM does not include such directives for multiple reasons. First, without strong understanding of the dependencies in the algorithm, use of such directives can easily lead to incorrect or unpredictable program behavior. Second, the directive-

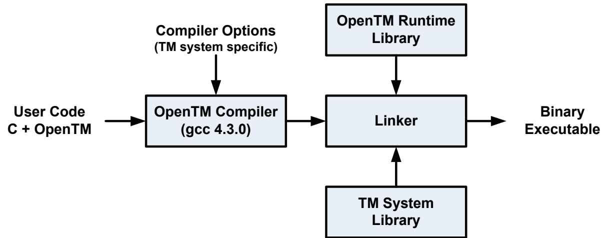

Figure 1. An overview of our first OpenTM implementation.

based syntax in OpenMP is static and quite limiting with respect to variables that it can describe (no array indices or pointer dereferences). Finally, the **`private`** and **`shared`** clauses in OpenMP already provide programmers with a mechanism to identify variables that are thread-private and participate in conflict detection between concurrent transactions. Nevertheless, in future versions of OpenTM, we will examine the inclusion of runtime routines for the *early release* [30].

Compilation Issues: For software TM systems, we must guarantee that any function called within an OpenTM transaction is instrumented for TM bookkeeping. If function pointers or partial compilation are used, the compiler may not be able to statically identify such functions. In this case, the programmer must use a **`tm_function`** directive to identify functions that may be called within a transaction [32]. The compiler is responsible for cloning the function in order to introduce the instrumentation and the code necessary to steer control to the proper clone at any point in time. For all TM systems, the compiler can freely reorder code within transactions, as it is guaranteed that any updates they perform will be made visible to the rest of the system atomically, when the transaction commits.

4 A First OpenTM Implementation

At the early stage of this work, we implemented OpenTM using the Cetus source-to-source translation framework [20]. While source-to-source translation allowed us to quickly develop a portable prototype, it has two important shortcomings. First, it limits the degree of compiler optimizations that can be applied to OpenTM code. For software and hybrid TM systems, optimizations such as eliminating redundant barriers or register checkpoints can have a significant impact on performance. In a full compiler framework, existing compilation passes can implement most of these optimizations [2, 32, 29], while, in a source-to-source translation framework, the type and scope of optimizations are limited. Second, source-to-source translation complicates code development, as debugging with tools like GDB must be done at the level of the low-level, translated output instead of the high-level, OpenTM code.

Our current OpenTM implementation is based on a full compiler. Specifically, we extended the GNU OpenMP (GOMP) environment [25], an OpenMP implementation for GCC 4.3.0. GOMP consists of four main components: the parser, the intermediate representation (IR), the code generator, and the runtime library. We extended the four components in the following manner to support OpenTM. The parser is extended to identify and validate the OpenTM pragmas and to generate the corresponding IRs or report compile-time errors. The GENERIC and GIMPLE intermediate languages were extended to represent the features of the low-level TM interface described in Table 2. Similar to OpenMP, the code generator performs all the code transformations for OpenTM, including the corresponding GIMPLE expansion and calls to runtime library routines before the Static Single Assignment (SSA) code generation stage [25].

The OpenTM implementation can generate code for hardware, software, and hybrid TM systems. Figure 1 summarizes the code generation process. The user code is written in C with OpenTM directives and does not have to be modified to target a different TM system. Command-line options guide the compiler to produce code for the right TM system. While the compiler targets a single low-level TM interface (see Table 2), it may use just a subset of the interface when targeting a specific TM system. For example, hardware TM systems do not require the use of read and write barriers such as **`TM_OpenWordForRead()`** and **`TM_OpenWordForWrite()`**. The compiler implements conventional optimizations (e.g., common subexpression elimination) and optimizations specific to the TM system [2]. Finally, the code is linked with the OpenTM runtime library and the TM system library that provides the implementation of the low-level interface.

We have successfully used the OpenTM implementation with three x86-based TM systems: a hardware TM system similar to TCC [21], the TL2 software TM system [11], and the SigTM hybrid TM system [6]. The implementation can target any TM system that supports the assumed low-level TM interface. GCC supports virtually all architectures, hence the implementation can target TM systems that do not use the x86 ISA. We are currently introducing the compiler optimizations for software TM systems [2, 29]. Note that, compared to the related work, the optimization opportunities can be somewhat reduced, as the OpenTM targets unmanaged languages such as C and C++, instead of a managed language like Java or C# [32].

Low-level TM Interface: Table 2 presents the low-level TM interface targeted by our OpenTM compiler. The interface describes the basic functionality in hardware, software, and hybrid TM systems. Columns H and S specify which functions are required by hardware and software TM sys-

Interface	Description	H	S
`void TM_BeginClosed(txDesc*)`	Make a checkpoint and start a closed-nested transaction.	√	√
`void TM_CommitClosed(txDesc*)`	Commit a closed-nested transaction.	√	√
`void TM_BeginOpen(txDesc*)`	Make a checkpoint and start an open-nested transaction.	√	√
`void TM_CommitOpen(txDesc*)`	Commit an open-nested transaction.	√	√
`bool TM_Validate(txDesc*)`	Validate the current transaction.	√	√
`void TM_Abort(txDesc*)`	Abort the current transaction.	√	√
`void TM_RegHd(txDesc*,type, callbackFn*,params)`	Register a software handler; `txDesc`: transaction descriptor, `type`: commit, violation, abort, `callbackFn`: function pointer to the handler.	√	√
`void TM_InvokeHd(txDesc*,type, callbackFn*,params)`	Invoke a software handler.	√	√
`void TM_SetCM(cmDesc)`	Set the contention management policy; `cmDesc`: contention management descriptor (type & parameters).	√	√
`cmDesc TM_GetCM()`	Return a contention management descriptor for the current policy.	√	√
`txDesc* TM_GetTxDesc(txDesc*)`	Get a transaction descriptor.	√	√
`uint32 TM_MonolithicReadWord(addr)`	Monolithic transactional read barrier.		√
`void TM_MonolithicWriteWord(addr, data)`	Monolithic transactional write barrier.		√
`void TM_OpenWordForRead(addr)`	Insert address in the transaction's read-set for conflict detection (decomposed read).		√
`uint32* TM_OpenWordForWrite(addr, data)`	Eager systems: insert into transaction's write-set and create undo log entry; Lazy systems: insert into transactions's write-set, allocate write-buffer entry and return its address.		√
`uint32 TM_ReadWordFromWB(addr)`	Lazy systems only: search write-buffer for address; if found, read value from write-buffer; otherwise return value in a regular memory location. Used when it is not certain at compile time, if a word has been written by this transaction or not.		√
`bool TM_ValidateWord(addr)`	Check the address for conflicts.		√

Table 2. The low-level TM interface targeted by the OpenTM compiler during code generation.

tems, respectively. Hybrid TM systems require the same functions as software TM systems. While these functions may be implemented differently across TM systems (e.g., TM systems with eager [26] vs. lazy [11] version management), they are sufficient to control the execution of memory transactions. Given an implementation of this interface, our OpenTM compiler can generate correct code for any TM system.

The first set of functions in Table 2 provides user-level threads to begin, commit, and abort transactions. These are common across all types of TM systems. We also require mechanisms to register and invoke software handlers on abort, commit, or a conflict [22]. The second set of functions provides the read and write barriers necessary for transactional bookkeeping for software and hybrid TM systems. We support both monolithic barriers (e.g., **`TM_MonolithicReadWord()`**) and decomposed barriers (e.g., **`TM_OpenWordForRead()`** and **`TM_ReadWordFromWB()`** or **`TM_ValidateWord()`**). While monolithic barriers are enough for correct execution, decomposed barriers reveal more opportunities for compiler optimization [2].

Runtime System: The runtime system in our current OpenTM implementation extends the GOMP runtime system with the small set of runtime library routines in Table 1. The system supports dynamic scheduling for transactional loops. It also provides basic support for contention management using a simple backoff policy. Conditional synchronization is currently implemented with immediate retries. Support for suspending threads on conditional synchronization is currently in progress. We are also planning to integrate the runtime system with online profiling tools in order to improve scheduling efficiency.

Feature	Description
Processors	1–16 x86 cores, in-order, single-issue
L1 Cache	64-KB, 32-byte line, private 4-way associative, 1 cycle latency
Network	256-bit bus, split transactions pipelined, MESI protocol
L2 Cache	8-MB, 32-byte line, shared 32-way associative, 12 cycle latency
Main Memory	100 cycles latency up to 8 outstanding transfers
Signatures	2048 bits per signature register

Table 3. Parameters for the simulated multi-core system.

5 Experimental Methodology

5.1 Environment

We use an execution-driven simulator that models multi-core systems with MESI coherence and support for hardware or hybrid TM systems. Table 3 summarizes the parameters for the simulated CMP architecture. All operations, except loads and stores, have a CPI of 1.0. However, all the details in the memory hierarchy timings, including contention and queueing events, are modeled. For the hardware TM system (HTM), the caches are enhanced with meta-data bits to support lazy version management and optimistic conflict detection [21]. For the hybrid TM system (SigTM), we use hardware signatures to accelerate conflict detection [6]. Data versioning is performed in software and, apart from the signatures, no further hardware modifications are needed. We also use the Sun TL2 software TM (STM) system [11], running on top of a conventional multi-core system without hardware enhancements for TM. The com-

piler output for the STM system can run on real machines. However, we run STM code on the simulated system to facilitate comparisons between the three TM systems (hardware, hybrid, and software).

5.2 Applications

We use four applications and one microbenchmark in our evaluation [6, 17]: *delaunay* implements Delaunay mesh generation for applications such as graphics rendering and PDE solvers; *genome* is a bioinformatics application and performs gene sequencing; *kmeans* is a data mining algorithm that clusters objects into `k` partitions based on some attributes; *vacation* is similar to the SPECjbb2000 benchmark and implements a travel reservation system powered by an in-memory database; *histogram* is a microbenchmark with multiple threads concurrently updating an array of histogram bins after some computational work. For *kmeans* and *vacation*, we use two input datasets that lead to different frequency of conflicts between concurrent transactions (low/high).

We developed the *OpenTM code* for all applications using coarse-grain transactions to execute concurrent tasks that operate on various shared data structures such as a graph and a tree. Parallel coding at this level is easy because the programmer does not have to understand or manually manage the inter-thread dependencies within the data structure code. The parallel code is very close to the sequential algorithm. The resulting runtime behavior is coarse-grain transactions account for most of the runtime. For comparison, we developed additional versions of the application code. We directly used the *low-level interface* available through the TM system library to develop another transactional version of the code with the similar parallelization. Due to the lower-level syntax, this version tends to be cumbersome to develop but can provide some opportunities for optimizations that are not exposed at the OpenTM level. Coarse-grain lock (CGL) versions are implemented by simply replacing transaction boundaries with lock acquisitions and releases. Fine-grain lock (FGL) versions are implemented by using lock primitives at the finest granularity possible in order to maximize concurrency. FGL code is quite difficult to debug, tune, and port, as the programmer is responsible for fine-grain concurrency management.

5.3 Example: Vacation Pseudocode in OpenTM

To show the simplicity of programming with OpenTM, Figure 2 presents the pseudocode of *vacation*. The programmer simply uses the **`transaction`** construct to specify that client requests should run in parallel as atomic transactions. There is no need to prove that the requests are independent or use low-level locks to manage the infrequent dependencies. This code achieves good performance because the underlying TM system implements optimistic concurrency, and conflicts between client requests

```
void client_run(args) {
  for(i=0; i<numOps; i++) {
    #pragma omp transaction
    { /* begin transaction */
      switch(action) {
        case MAKE_RESERVATION:
          do_reservation();
        case DELETE_CUSTOMER:
          do_delete_customer();
        ...}
    } /* end transaction */ }}
void main() {
  #pragma omp parallel {
    client_run(args);}}
```

Figure 2. OpenTM pseudocode for vacation.

Application	File	# of extra C lines			
		FGL	LTM-H	LTM-S	OTM
delaunay	cavity.c	43	0	0	0
	delaunay.c	16	18	24	22
	mesh.c	0	0	4	0
	worklist.c	0	0	10	0
genome	genome.c	8	8	10	8
	hashtable.c	3	0	6	0
	sequencer.c	25	32	58	11
	table.c	3	0	0	0
kmeans	normal.c	25	23	31	11
vacation	client.c	0	2	2	2
	customer.c	0	0	1	0
	manager.c	0	0	7	0
	rbtree.c	11	0	105	0
	reservation.c	0	0	20	0
	vacation.c	8	8	10	8

Table 4. Number of extra lines of C code needed to parallelize each application using fine-grain locks, the low-level TM interface for hardware and software TM systems, and OpenTM.

are not particularly common. Using coarse-grain locks requires similar programming effort, but does not scale due to serialization. Better scalability can be achieved by using fine-grain locks, but at the expense of programmability. The user must manage the fine-grain locks as trees are traversed, updated, and potentially rebalanced in a manner that avoids deadlocks.

6 Evaluation

6.1 Programmability

Quantifying programming effort is a difficult task that requires extensive user studies. Nevertheless, Table 4 shows the number of extra lines of C code needed to parallelize each application using fine-grain locks, the low-level TM interface, and OpenTM constructs as one indication of coding complexity. For the low-level TM interface, we present two sets of results: for hardware TM systems where the programmer must define transaction boundaries, and for software TM systems where the programmer must also introduce read/write barriers. Table 4 does not report results for

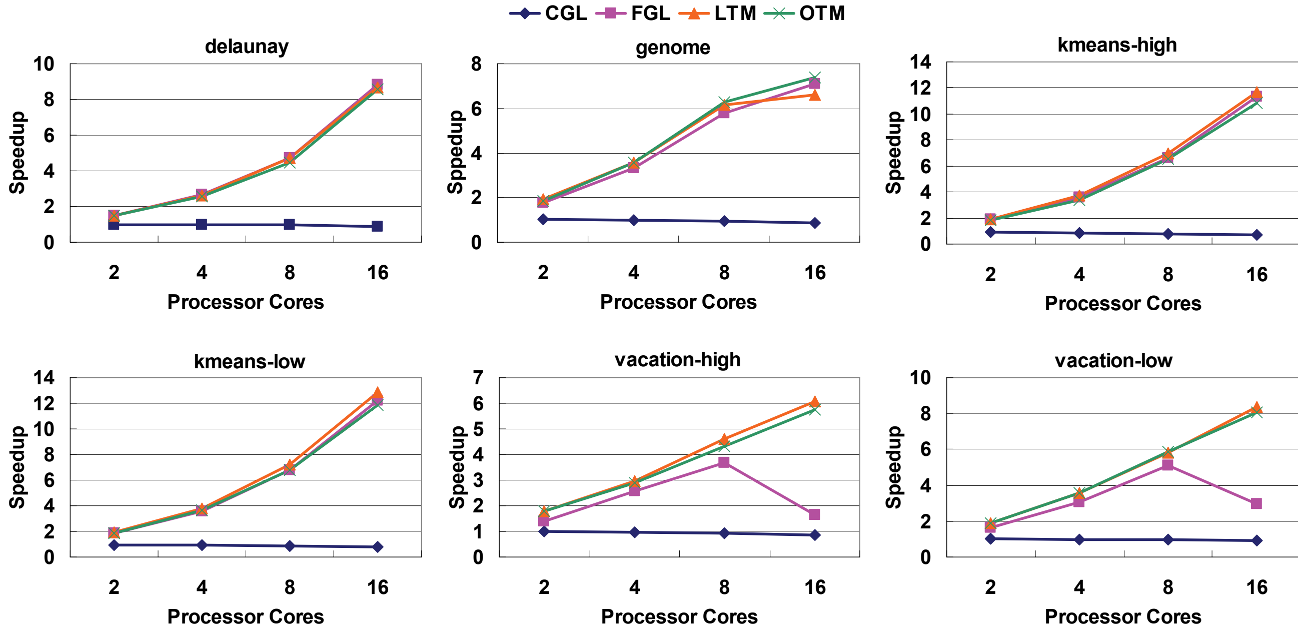

Figure 3. Application speedups with coarse-grain locks (CGL), fine-grain locks (FGL), low-level TM interface (LTM), and OpenTM (OTM).

coarse-grain locking code, as they are essentially identical to those of low-level TM code for a hardware TM system.

Compared to the OpenTM code, FGL requires significantly more programming effort, as the user must manually orchestrate fine-grain accesses to shared states apart from identifying work-sharing opportunities. For example, the FGL code for *delaunay* requires optimistic parallelization and places the burden on the user to detect and handle conflicts between concurrent cavity expansions using fine-grain locks [17]. Apart from the extra code, the programmer must make sure that there is no deadlock and no livelock, and that contention is managed in a reasonable manner. In OpenTM code, the programmer simply identifies work-sharing and memory transactions. Concurrency management is the responsibility of the system, including conflict detection, contention management, and dynamic scheduling.

OpenTM has also advantages over directly using a low-level TM interface, For software TM systems, use of the low-level interface requires manual instrumentation and optimization of load and store accesses within a transaction. As it is most obvious in the case of *vacation*, instrumentation code leads to significant code size increase and can be tricky to handle. If the programmer introduces redundant barriers, there can be a significant impact on performance [2]. If the programmer misses a barrier, the program behavior can be incorrect or unpredictable depending on the input dataset and runtime environment. OpenTM eliminates this tradeoff by automatically introducing and optimizing barriers for shared variables during compilation. We found the OpenTM approach advantageous even when targeting a hardware TM system. While read/write barriers are not needed in this case, significant code transformation is often required to distribute loop iterations and implement dynamic scheduling (e.g., *genome* and *kmeans*). The low-level code is also difficult to change to make any optimizations such as changing the chunk size or transaction size.

As is the case with OpenMP, the OpenTM code requires simple, high-level annotations for parallelism and memory transactions. It is relatively easy to understand, maintain, and reuse without requiring expert programmers or error-prone programming techniques.

6.2 Performance Comparison

Figure 3 presents the speedups for the four code versions of each application, as we scale the number of processors from 2 to 16 (higher is better). Speedup is relative to the sequential version of each application. The two TM versions assume hardware TM support. To provide further insights, Figure 4 presents execution times with 16 processors, normalized to the sequential execution time. Each bar is broken down into time executing useful instructions, time for cache misses, idle and synchronization time, transaction commit time, and time spent on aborted transactions (violation).

OpenTM code compares favorably to lock-based code. As expected, CGL code does not scale due to the serialization of coarse-grain locks. FGL code exhibits excellent scaling for three applications. For *vacation*, FGL is penal-

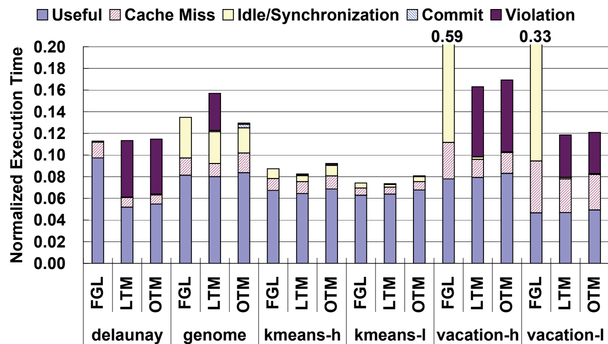

Figure 4. Normalized execution time with 16 processors.

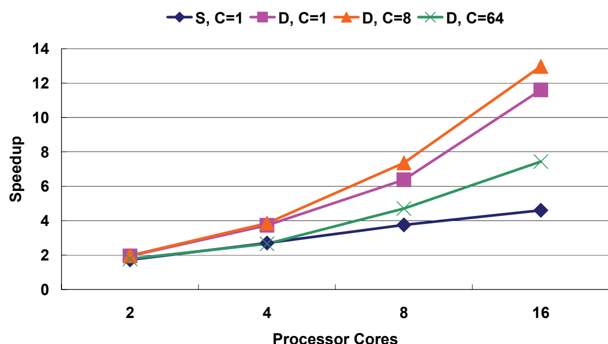

Figure 5. Speedups for the histogram with 16 processors and static (S) or dynamic (D) scheduling. C represents the chunk size.

ized by the overhead of acquiring and releasing locks, as it searches and updates tree structures. The OpenTM code scales well across all programs. For some applications (e.g., *delaunay*), FGL code is marginally faster than OpenTM code due to the overhead of aborted transactions when optimistic concurrency fails. On the other hand, OpenTM code is marginally (e.g., *genome*) or significantly (e.g., *vacation*) faster than FGL code as it avoids the overhead of fine-grain lock management.

Figure 3 and 4 also capture the performance differences between OpenTM code and low-level TM code. Since both versions utilize the hardware support for TM, they perform similarly for all system scales. Using the simpler, higher-level syntax of OpenTM does not lead to performance issues. For *genome*, OpenTM code is faster due to better scheduling and contention management using the automated approach in OpenTM. Similar optimizations are difficult to introduce and tune in the low-level code.

6.3 Runtime System

Dynamic Scheduling: To demonstrate the usefulness of

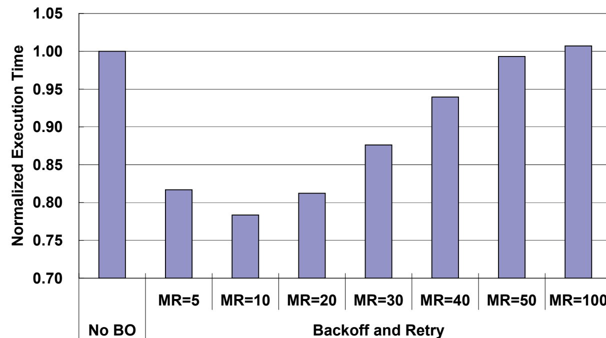

Figure 6. Normalized execution time for the histogram with 16 processors using various contention management configurations.

the dynamic scheduling capabilities in the OpenTM runtime, we use the *histogram* microbenchmark. As each histogram bin is updated after a varying amount of computational work, there can be a significant imbalance. Figure 5 presents the speedups with up to 16 processors. The *static* version uses the `static` clause to statically schedule loop iterations in an interleaved fashion with a chunk size of 1. The *dynamic* versions use the `dynamic` clause to dynamically schedule loop iterations with chunk size of 1 to 64 iterations. For simplicity, we set the transaction size equal to the chunk size. The dynamically scheduled executions scale significantly better than the statically scheduled one, as they eliminate work imbalance. The differences between the dynamic versions are due to the tradeoff between eliminating the scheduling overhead with smaller chunk size (e.g., frequently accessing the global work queue) and allowing for some imbalance and more frequent transaction aborts with a larger chunk size. Note that changing the scheduling policy, the chunk size, and the transaction size are simple in OpenTM and do not require global code changes, as is the case when using lower-level interfaces.

Contention Management: We also use the *histogram* microbenchmark to showcase the usefulness of contention management in OpenTM. Figure 6 presents normalized execution time with 16 processors and various contention management configurations. The left-most bar represents the execution time without any contention management. As soon as a transaction aborts, there is an immediate retry. The remaining bars represent execution time with a simple contention management policy. When a transaction aborts, it uses linear backoff. If a transaction aborts *MR* times, it requests to be executed as a high priority transaction that wins conflicts with other concurrent transactions. Backoff reduces the amount of wasted work when multiple threads operate on the same data. The priority mechanism provides fairness across threads, as long transactions cannot starve

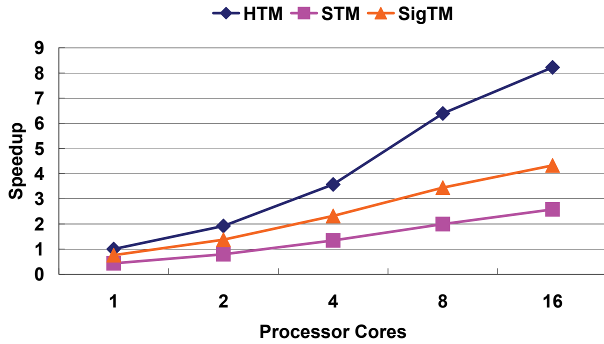

Figure 7. Speedups for the histogram OpenTM code running on the hardware, software, and hybrid TM systems.

due to conflicts with smaller transactions [5].

Figure 6 shows that there is a 22% reduction in execution time using contention management (MR=10). It also demonstrates the importance of tuning the contention management policy. With low MR values, the priority mechanism is used frequently, causing some performance loss due to serialization. In the hardware TM system we used, when a transaction uses the priority mechanism, no other transaction can commit, even if there is no conflict detected. When MR is high, there is performance loss, as long transactions are repeated multiple times, causing imbalance in the system. Overall, contention management can be very important for robust performance in TM systems. OpenTM allows users to provide high-level hints in order to guide the runtime system in addressing contention challenges.

6.4 Code Generation for Software TM Systems

The results thus far are from the execution of OpenTM code on the hardware TM system. However, our OpenTM compiler produces code for software and hybrid TM systems as well. Figure 7 presents the speedups achieved with the OpenTM code for *histogram* on all three TM systems available to us. As expected, the hardware TM system is the fastest, as it eliminates all software overhead for transactional bookkeeping. The hybrid TM system (SigTM) addresses some of the shortcomings of the software system and provides a 2$\times$ performance improvement over software-only TM. While a detailed comparison between the three TM implementation approaches is beyond the scope of this paper (see [6] for details), Figure 7 demonstrates the OpenTM code can be mapped to a variety of TM systems. With proper compiler optimizations, the OpenTM environment can provide a scalable and portable way to build efficient parallel programs.

7 Related Work

High-level parallel programming is a topic with a long history of research and development efforts. This paper is most related to the shared-memory approach of OpenMP and its predecessors [1].

Recently, there has been significant research on high-level programming environments for transactional memory or similar parallel systems. Wang et al. proposed extensions to the C programming language and an optimizing C compiler to support software TM [32]. Our proposal differs from their work by building on top of OpenMP. Instead of focusing only on software TM issues, we propose an integrated system that includes TM features along with language and runtime constructs to express and manage parallelism (e.g., work-sharing and dynamic scheduling). von Praun et al. proposed the IPOT programming environment that uses ordered transactions to support speculative parallelization [31]. OpenTM supports both ordered and unordered transactions, as we recognize the non-blocking synchronization is as important as thread-level speculation. IPOT also lacks full programming constructs (e.g., work-sharing and advanced TM features) that help programmers develop and optimize transactional applications.

In parallel with this work, Milovanovic et al. proposed to extend OpenMP to support TM [23]. Their proposal defines a transaction construct similar to the one in OpenTM and includes an implementation based on source-to-source translation. In addition to the transaction construct, OpenTM defines constructs for features such as conditional synchronization, nesting, transactional handlers, and contention management. Moreover, OpenTM has been implemented using a compiler-based approach. Nevertheless, it is reasonable to merge the two proposals into a unified set of OpenMP extensions for TM programming.

Several researchers have investigated TM support for managed languages such as Java [15, 2, 7]. Such environments use similar constructs for TM programming but exploit the dynamic code generation, type safety, and object-oriented features of such languages. The OpenTM infrastructure targets unmanaged languages such as C and C++, focusing on the synergy between high-level OpenMP programming and transactional mechanisms for non-blocking synchronization and speculative parallelization.

8 Conclusions

This paper presented OpenTM, a high-level API for parallel programming with transactions. OpenTM extends the familiar OpenMP environment to support both non-blocking synchronization and speculative parallelization by using memory transactions. We also presented a first implementation of OpenTM that is based on the GCC compiler. The system can produce code for hardware, software, and hybrid TM systems and includes a runtime system with

dynamic scheduling and contention management features. We evaluated the programmability and performance of the OpenTM environment and showed that it delivers good performance with simple and portable high-level code. Overall, OpenTM provides a practical and efficient TM programming environment within the familiar scope of OpenMP.

We are currently improving the compiler and runtime system for OpenTM to introduce TM-specific optimizations, improve scheduling for conditional synchronization, and add further contention management schemes. We are also investigating further language extensions for features such as nested parallelism and relaxed transactional semantics for cognitive applications. Finally, we intend to open-source the OpenTM environment in order to encourage further application development and experimentation with transactional memory.

Acknowledgements

We would like to thank the anonymous reviewers for their feedback at various stages of this work. We also want to thank Sun Microsystems for making the TL2 code available. This work was supported by NSF Career Award number 0546060, NSF grant number 0444470, and FCRP contract 2003-CT-888. Woongki Baek is supported by an STMicroelectronics Stanford Graduate Fellowship and a Samsung Scholarship.

References

[1] The OpenMP Application Program Interface Specification, version 2.5. `http://www.openmp.org`, May 2005.

[2] A.-R. Adl-Tabatabai, B. Lewis, et al. Compiler and Runtime Support for Efficient Software Transactional Memory. In *the Proceedings of the 2006 Conf. on Programming Language Design and Implementation*, June 2006.

[3] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996.

[4] C. S. Ananian, K. Asanović, et al. Unbounded Transactional Memory. In *the Proceedings of the 11th Intl. Symp on High-Performance Computer Architecture*, Feb. 2005.

[5] J. Bobba, K. E. Moore, et al. Performance pathologies in hardware transactional memory. In *the Proceedings of the 34th Intl. Symp. on Computer Architecture*, pages 81–91, New York, NY, USA, 2007. ACM Press.

[6] C. Cao Minh, M. Trautmann, et al. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *the Proceedings of the 34th Intl. Symp. on Computer Architecture*. June 2007.

[7] B. D. Carlstrom, A. McDonald, et al. The Atomos Transactional Programming Language. In *the Proceedings of the Conf. on Programming Language Design and Implementation*, June 2006.

[8] W. Chuang, S. Narayanasamy, et al. Unbounded Page-Based Transactional Memory. In *the Proceedings of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. Oct. 2006.

[9] J. Chung, C. Cao Minh, et al. Tradeoffs in Transactional Memory Virtualization. In *the Proceedings of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. Oct. 2006.

[10] J. Chung, H. Chafi, et al. The Common Case Transactional Behavior of Multithreaded Programs. In *the Proceedings of the 12th Intl. Conf. on High-Performance Computer Architecture*, Feb. 2006.

[11] D. Dice, O. Shalev, et al. Transactional Locking II. In *the Proceedings of the 20th Intl. Symp. on Distributed Computing*, Sept. 2006.

[12] R. Guerraoui, M. Herlihy, et al. Robust Contention Management in Software Transactional Memory. In *the OOPSLA Workshop on Synchronization and Concurrency in Object-Oriented Languages*. Oct. 2005.

[13] T. Harris. Exceptions and Side-effects in Atomic Blocks. In *the PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2004.

[14] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *the Proceedings of the 18th Conf. on Object-oriented Programing, Systems, Languages, and Applications*, Oct. 2003.

[15] T. Harris, S. Marlow, et al. Composable Memory Transactions. In *the Proceedings of the Symp. on Principles and Practice of Parallel Programming*, July 2005.

[16] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *the Proceedings of the 20th Intl. Symp. on Computer Architecture*, May 1993.

[17] M. Kulkarni, K. Pingali, et al. Optimistic Parallelism Requires Abstractions. In *the Proceedings of the Conf. on Programming Language Design and Implementation*, June 2007.

[18] S. Kumar, M. Chu, et al. Hybrid Transactional Memory. In *the Proceedings of the 11th Symp. on Principles and Practice of Parallel Programming (PPoPP)*, New York, NY, Mar. 2006.

[19] J. Larus and R. Rajwar. *Transactional Memory*. Morgan Claypool Synthesis Series, 2007.

[20] S. I. Lee, T. A. Johnson, et al. Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation. In *the Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, Oct. 2003.

[21] A. McDonald, J. Chung, et al. Characterization of TCC on Chip-Multiprocessors. In *the Proceedings of the 14th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2005.

[22] A. McDonald, J. Chung, et al. Architectural Semantics for Practical Transactional Memory. In *the Proceedings of the 33rd Intl. Symp. on Computer Architecture*, June 2006.

[23] M. Milovanovic, R. Ferrer, et al. Transactional Memory and OpenMP. In *the Proceedings of the Intl. Workshop on OpenMP*, June 2007.

[24] K. E. Moore, J. Bobba, et al. LogTM: Log-Based Transactional Memory. In *the Proceedings of the 12th Intl. Conf. on High-Performance Computer Architecture*, Feb. 2006.

[25] D. Novillo. Openmp and automatic parallelization in gcc. In *the Proceedings of the GCC Developers' Summit*, June 2006.

[26] B. Saha, A.-R. Adl-Tabatabai, et al. A High Performance Software Transactional Memory System For A Multi-Core Runtime. In *the Proceedings of the 11th Symp. on Principles and Practice of Parallel Programming*, Mar. 2006.

[27] B. Saha, A.-R. Adl-Tabatabai, et al. Architectural Support for Software Transactional Memory. In *the Proceedings of the 39th Intl. Symp. on Microarchitecture (MICRO)*, Dec. 2006.

[28] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *the Proceedings of the 24th Symp. on Principles of Distributed Computing*, July 2005.

[29] T. Shpeisman, V. Menon, et al. Enforcing Isolation and Ordering in STM. In *the Proceedings of the Conf. on Programming Language Design and Implementation*, Mar. 2007.

[30] T. Skare and C. Kozyrakis. Early release: Friend or foe? In *the PLDI Workshop on Transactional Memory Workloads*. Jun 2006.

[31] C. von Praun, L. Ceze, et al. Implicit Parallelism with Ordered Transactions. In *the Proceedings of the 12th Symp. on Principles and Practice of Parallel Programming*, Mar. 2007.

[32] C. Wang, W.-Y. Chen, et al. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *the Proceedings of the Intl. Symp. on Code Generation and Optimization*, Mar. 2007.