

# Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language

Cheng Wang, <sup>‡</sup>Wei-Yu Chen, Youfeng Wu, Bratin Saha, Ali-Reza Adl-Tabatabai

Programming System Lab  
Microprocessor Technology Labs  
Intel Corporation  
2200 Mission College Blvd  
Santa Clara, CA 95053

<sup>‡</sup> Computer Science Division  
University of California, Berkeley  
575 Soda Hall  
wychen@cs.berkeley.edu

{cheng.w.wang,youfeng.wu,bratin.saha,ali-raza.adl-tabatabai}@intel.com

## Abstract

*Transactional memory offers significant advantages for concurrency control compared to locks. This paper presents the design and implementation of transactional memory constructs in an unmanaged language. Unmanaged languages pose a unique set of challenges to transactional memory constructs – for example, lack of type and memory safety, use of function pointers, aliasing of local variables, and others. This paper describes novel compiler and runtime mechanisms that address these challenges and optimize the performance of transactions in an unmanaged environment. We have implemented these mechanisms in a production-quality C compiler and a high-performance software transactional memory runtime. We measure the effectiveness of these optimizations and compare the performance of lock-based versus transaction-based programming on a set of concurrent data structures and the SPLASH-2 benchmark suite. On a 16 processor SMP system, the transaction-based version of the SPLASH-2 benchmarks scales much better than the coarse-grain locking version and performs comparably to the fine-grain locking version. Compiler optimizations significantly reduce the overheads of transactional memory so that, on a single thread, the transaction-based version incurs only about 6.4% overhead compared to the lock-based version for the SPLASH-2 benchmark suite. Thus, our system is the first to demonstrate that transactions integrate well with an unmanaged language, and can perform as well as fine-grain locking while providing the programming ease of coarse-grain locking even on an unmanaged environment.*

## 1. Introduction

Chip multiprocessors (CMPs) are now part of mainstream computing with some processors supporting up to 32 hardware threads [8]. Applications must now become concurrent to leverage the computing power of CMPs. Programmers currently use locks to enforce mutual exclusion in concurrent applications. Locks, however, cause a number of software engineering problems – such as deadlock, non-scalable composition, priority inversion, and others – that make it difficult for

developers to compose scalable applications out of software components.

Transactional memory (TM) [14] eliminates many of the problems associated with locks and enables developers to compose scalable applications safely. Recent work [1] [2] [6] [11] [13] [10] has added first-class TM constructs (in the form of an atomic block, or transaction) to new or existing languages. A TM language construct not only provides syntactic convenience and some static safety guarantees, but also enables the compiler to optimize the overheads of TM.

So far, the work on TM language constructs has focused exclusively on managed languages such as Java, Haskell, and ML [1] [13] [10] [28]. Such languages run within a managed runtime environment (such as a Java virtual machine) and provide advanced language features such as type safety, object orientation, automatic memory management, first-class exception handling, and just-in-time (JIT) compilation. These properties facilitate integrating TM into the language; for example, the TM implementation can leverage the exception handling mechanisms to implement control transfer for transaction aborts [1], and since exceptions are integrated into the language, the runtime can validate a transaction to check the transaction's consistency before propagating an exception out of a transaction. Managed environments also facilitate compiler optimizations that target the overheads of TM; for example, the JIT compiler can eliminate redundant TM read and write barriers applied to the same object and can lazily clone methods invoked inside a transaction, avoiding TM instrumentation overhead on code that is not executed in a transaction [1].

Supporting transactions in an unmanaged language such as C and C++ is important because of the huge investment in existing unmanaged code, but supporting transactions in an unmanaged language has a unique set of problems that make it more challenging than managed code. C compilers, for example, generate code statically and so must either generate TM instrumentation for all functions, which will slow down the application significantly, or figure out statically the set of functions to clone, which is challenging in the presence of separate compilation and function pointers. Further, the lack of

type safety and the presence of unsafe pointer arithmetic force the use of cache-line rather than object granularity transactional memory, and unlike type-safe environments, require the TM implementation to check transaction consistency aggressively at each transactional memory access. Both of these restrictions make it more challenging to optimize the overheads of TM in an unmanaged environment.

To illustrate some of the challenges of unmanaged code, consider the code in Figure 1 adapted from the function *get\_task()* in the *radiosity* benchmark of the SPLASH-2 suite. In a transactional setting with read concurrency, two threads T1 and T2 can simultaneously start executing the code block, find that the free list is non-null (*tq->free* is not zero), read the head of the free list (*temp = tq->free*), and find that the next field (*temp->next*) is non-null at L1. One of the threads (say T2) may then get delayed, while the other thread (say T1) exits the *for* loop, sets *temp->next* to null (L4), and commits the transaction. In essence, T1 dequeues the head of the free list, adds it to its local free list (*loc\_free*) and moves the head of the free list (L3). T2 can then continue execution from L1, but it now has an inconsistent value of *temp->next*. In particular, T2 will incur a fault when it executes the assignment *temp = temp->next* and checks the condition *temp->next*. In general, an inconsistent transaction can also corrupt the runtime's state by storing through an inconsistent pointer value.

```

atomic {
  if( tq->free ) {
    /* Scan the free list */
    for(temp = tq->free;
        temp->next && ...;
L1:      temp = temp->next
L2:    );

    task_struct[p_id].loc_free = tq->free;
L3:  tq->free = temp->next;
L4:  temp->next = NULL;
    ...
  } /* end atomic */
temp = task_struct[p_id].loc_free;
/* process temp */
task_struct[p_id].loc_free = temp->next;
L5: temp->next = NULL;

```

**Figure 1: Consistency issues for unmanaged transactions**

The TM implementation needs to ensure that any exceptions or error conditions generated due to an inconsistency are not exposed to the application or do not compromise the integrity of the runtime. Managed TM implementations handle this by leveraging type-safety and exception handling [1] [10]. Type-safety ensures that critical runtime structures are not corrupted, while the managed runtime validates transactions on an exception – for example, in Figure 1, T2 would throw a null pointer exception, and the resulting validation would abort the transaction.

The inconsistency in Figure 1 arises from using optimistic read concurrency and delayed validations. One option is to use pessimistic read concurrency, and after every read, check that a transaction is valid; however, prior work [29] has shown that this is an order of magnitude less efficient than using optimistic read concurrency. So we would like to stick with optimistic read concurrency, but ensure that temporary inconsistencies don't lead to spurious exceptions (which manifest as OS signals on Unix platforms), or failures resulting from critical STM data structures getting corrupted. Note that a write buffering STM implementation (with optimistic read concurrency) can also lead to inconsistent behavior and spurious signals. In an unmanaged environment, even if the TM implementation registers a signal handler to catch spurious signals, other parts of the application may register other handlers and override the TM handler. Moreover, there is no guarantee on how the different handlers will get invoked. Therefore, in an unmanaged environment, the TM implementation must ensure that inconsistencies are detected promptly and do not lead to spurious errors.

This paper focuses on adding transactional memory constructs to unmanaged languages such as C/C++. We propose new C language extensions that support transactions. We present a high-performance TM stack comprising a production-quality optimizing C compiler [30] and a high-performance software transactional memory (STM) runtime [29]. This paper makes the following novel contributions:

It is the first to introduce comprehensive transactional memory constructs to the C programming language. Prior work has focused mostly on constructs for managed languages. We introduce new constructs in the form of C-language pragmas. These constructs allow a programmer to declare blocks that execute atomically, and to annotate functions that can be called inside a transaction. We also provide an intrinsic function that allows a programmer to (partially) rollback a (nested) transaction.

It is the first to support transactions in a production-quality optimizing C compiler. We show how to generate code for the new TM constructs and present compiler optimizations that target the overheads of STM while accounting for the restrictions imposed by an unmanaged environment. We introduce a new mechanism to resolve the target of an indirect function call in the presence of function cloning and interface with legacy library functions.

It presents a novel STM algorithm and API that supports the requirements of an optimizing compiler in an unmanaged environment. We extend a high-performance STM runtime with a time-stamp mechanism that preserves a transaction's consistency even in the presence of optimistic read concurrency, so that an application does

not encounter any spurious errors. Prior STM implementations did not deal with aliasing of local variables, a common idiom in C programs. We show how local variable aliasing can corrupt an STM system, and present an algorithm to deal with local variable aliasing. Finally, we show how a STM using optimistic readers can handle programming idioms such as privatization.

We evaluate our system on a 16-way SMP system using the SPLASH-2 benchmarks and a set of concurrent data structures. Our measurements show that in a single-thread execution, transactions introduce about 6.4% overhead compared to locks, and in multi-threaded execution, transactions scale comparably to fine-grain locking. Moreover, our compiler optimizations reduce transaction overhead significantly.

The rest of this paper is organized as follows. Section 2 describes our new TM language constructs for C. Section 3 describes how the compiler maps these constructs to functions in the STM runtime. Section 4 describes the STM runtime algorithms. Section 5 describes our compiler optimizations. Section 6 presents measurements. Section 7 discusses related work, and Section 8 concludes the paper and discusses future directions.

## 2. TM language constructs

We introduce 3 constructs for supporting transactions. The first construct, `#pragma tm_atomic`, demarcates a code block that executes atomically; e.g. in Figure 2, statements `stmt1` and `stmt2` execute as a transaction. The statements in a transaction can include function calls. A transaction can be nested inside another transaction. The effects of a nested transaction are visible only when the outermost transaction commits, thus our system implements *closed nesting* [16]. A data conflict rolls back to the outermost level and re-executes the transaction.

```
#pragma tm_atomic
{
    stmt1;
    stmt2;
}
```

Figure 2: Atomic pragma

The second construct, the `tm_abort()` intrinsic function, rolls back the state to what it was on entry to the innermost transaction by undoing the stores, and re-executes the innermost transaction. This is similar to retry in [13]. For example, in Figure 3, the abort rolls back the updates in `stmt2` and re-executes this statement.

```
#pragma tm_atomic
{
    stmt1;
    #pragma tm_atomic
    {
        stmt2;
        tm_abort();
    }
}
```

Figure 3: Nested transactions and abort

The third construct, `#pragma tm_function`, marks functions that can be called inside transactions; we refer

to such functions as *transactional functions*. The programmer annotates transactional function declarations in source and header files using this pragma. The programmer also needs to annotate function pointer declarations with the `tm_function` pragma if they are to be called inside transactions. The compiler allows a transaction to call only transactional functions and allows transactional functions to call only other transactional functions; for example, in Figure 4, the compiler allows the call to function `foo` inside the transaction but generates a compile-time error for the call to function `bar` because the programmer annotates only the declaration of `foo`. The compiler also checks that only a transactional function is assigned to a transactional function pointer.

Even though the `tm_function` pragma allows the compiler to statically determine whether an indirect call goes to a transactional function, the compiler still generates code that checks at runtime that an indirect call inside a transaction does indeed invoke a transactional function. If a transaction indirectly calls a non-transactional function, the generated code will abort the transaction and raise a runtime exception. This protects against unsafe casting or incompatible binaries, improving the overall reliability of the system and the integrity of the transactions. As an on-going research, we are working on an option to invoke a dynamic binary translator to convert the binary code to a transactional version [35].

```
#pragma tm_function
int foo (int);

int bar(int);
...
#pragma tm_atomic
{
    foo(3);
    bar(10); // compiler flags an error
}
```

Figure 4: TM annotated functions

For each transactional function, the compiler generates a *transactional clone* called only when inside transactions. The compiler mangles the names of these clones to distinguish them from the regular non-transactional versions of functions. When generating code for a transaction or a transactional clone, the compiler knows statically that it is generating code that will execute only inside a transaction (*transactional code*) and hence generates code specialized to contain the necessary calls to the STM runtime. The `tm_function` pragma bounds code duplication since the compiler generates transactional code only for a subset of functions.

Like other software transactional memory systems, our system implements weak atomicity, and hence it does not enforce isolation between a transaction and non-transactional code. Some programming idioms can lead to shared data being accessed outside of transactions – for example, double-checked locking. In Figure 1, the access

to *temp->next* (L5) may potentially conflict with a concurrent access to the same field by another thread executing inside the transaction – for example, two threads T1 and T2 can simultaneously execute till L1, then thread T2 may get delayed while T1 continues and executes L5. When thread T2 resumes execution, it will incur a segmentation violation due to the write (L5) from T1. Notice that for shared memory allocated inside a transaction, e.g. via `malloc()`, the use of the memory outside the transaction has the similar problem. If shared data is accessed outside a transaction, then the user must properly guard such accesses with transactions. The timestamp-based validation in our runtime can guarantee isolations as long as all the shared memory accesses are in transactions. The compiler could also automatically convert such memory accesses as single instruction transactions, but this may add significant overhead since the compiler would have to be conservative in detecting shared data (due to aliasing). When our compiler sees such single instruction transactional code blocks (also referred to as mini-transactions), it optimizes these blocks heavily and generates very efficient code. In fact, these mini-transactions are executed without any calls into the STM runtime.

### 3. Generating code for TM constructs

When generating code for the TM language constructs, the compiler inserts calls to the runtime STM API (Figure 5). The STM functions take an explicit transaction descriptor object obtained via the `stmGetTxnDesc` function. The transaction descriptor is a thread-local object that maintains the state of a transaction. Exposing the transaction descriptor allows the compiler to eliminate redundant accesses to thread-local storage.

```
TxnDesc* stmGetTxnDesc();
uint32  stmStart(TxnDesc*, TxnMemento*);
uint32  stmStartNested(TxnDesc*, TxnMemento*);
void    stmCommit(TxnDesc*);
void    stmCommitNested(TxnDesc*);
void    stmUserAbort(TxnDesc*);
void    stmAbort(TxnDesc*);
uint32  stmValidate(TxnDesc*);
uint32* stmComputeTxnRec(uint32* addr);
uint32  stmRead(TxnDesc*, uint32* txnRec);
void    stmWrite(TxnDesc*, uint32* txnRec);
void    stmCheckRead(TxnDesc*, uint32* txnRec,
                    uint32 version);
void    stmHandleContention(TxnDesc*);
Void    stmUndoLog(TxnDesc*, uint32* addr,
                  uint32 size);
```

Figure 5: STM API for compiled code

To start and commit a transaction, the compiler inserts calls to the `stmStart` and `stmCommit` functions at each outermost transaction’s start and end, respectively. For a transaction nested inside another transaction, the compiler similarly inserts calls to the `stmStartNested` and `stmCommitNested` functions. We use mementos [1] to snapshot the transaction logs and support partial rollback for nested transactions.

Inside a transaction, the compiler generates a call to an STM read/write barrier function for each load/store that may access shared memory. These barrier functions perform the required STM book-keeping to ensure atomicity and consistency of the transaction. Since we use cache-line based transactional memory, the compiler uses the `stmComputeTxnRec` function to compute the transaction record corresponding to a memory location, and passes this to the barrier functions. Exposing the `stmComputeTxnRec` to the compiler allows it to eliminate redundant computations of the transaction record. The `stmCheckRead` function is used to ensure that individual reads are consistent (Section 4), while the `stmValidate` function ensures that a transaction is consistent at the point the function is called.

For direct function calls inside transactional code, the compiler may generate calls to the transactional clone by using its mangled name. Calls through function pointers, however, are more complicated since the function pointer can point to any transactional function, and the compiler can not statically determine the call target. A function pointer always points to the non-transactional version of a function but we insert a special “noop-marker” in the beginning of the normal version of a transactional function. Since a non-transactional function does not have this special marker, a quick runtime check can determine whether a function pointer has a corresponding transactional clone. Although it is possible to place the special marker before the function, this, however, can cause an accidental match with the marker bits as compiler and tools can place arbitrary data before a function. We also insert the address to the transactional clone right before the beginning of a transactional function. If a function pointer points to a transactional function, the pointer is adjusted and the transactional clone is called. Otherwise, the generated code calls an internal runtime function (`handleBinary()`) that aborts the transaction and raises a runtime exception. This improves the overall robustness of our system and provides an option in the future to handle non-transactional function binaries dynamically through other means such as with a lightweight binary translator.

Figure 6 shows two indirect calls with function pointers (a). The transaction first calls the function `foo()` through the function pointer `fptr1`. The code generated for the call to `foo()` inside the transaction (b) first checks that the first instruction at the function entry is the noop-marker, and then calls the transactional clone of `foo()` pointed to by `(fptr1 - 4)`. The transaction next calls the function `qsort()` through the function pointer `fptr2`. Since `qsort()` is a library function whose first instruction is not the noop-marker, a runtime function `handleBinary` is called to deal with this non-transactional function. Outside the transaction, the call to `foo()` is simply generated as a call to `foo_1()`, a function that is equivalent

to `foo()` except it skips the first noop-marker instruction (c). Thus only indirect calls suffer an additional one instruction overhead incurred by the noop marker.

```

// declared to be transactional function
#pragma tm_function
int foo(int);

...
fptr1 = foo;
fptr2 = qsort; // lib

#pragma tm_atomic
{
    (*fptr1)(3);
    (*fptr2)(...)
}

    (a) Source code

if (*fptr1 == "noop-marker")
    call ** (fptr1 - 4)
else
    handleBinary(fptr1);
if (*fptr2 == "noop-marker")
    call ** (fptr2 - 4)
else
    handleBinary(fptr2);
    (b) Pseudo code generated for the
        Transaction

    Ox3723456 # address of transactional clone
<foo>:
    movl %eax, 0xmagic # noop marker
# actual function here
<foo_1>:
    Push ebp;
    (c) Compiler generated code for foo()

```

Figure 6: Handling function pointers inside transactions

#### 4. STM runtime

Our STM algorithm builds on the high-performance McRT-STM algorithms [29], extending them with a timestamp mechanism. We use cache-line granularity conflict detection and implement strict two-phase locking for writes. Writes update values in-place and generate undo log entries. The compiler specifies the number of bytes being written so that the runtime undoes only the appropriate bytes on a conflict. The STM uses the same read set, write set, and undo log structures as McRT-STM. Unlike the original McRT-STM, however, the STM (1) uses timestamps rather than versions to implement optimistic concurrency control for reads, (2) maintains a global timestamp value that each transaction increments on completion (commit or abort), and (3) extends the transaction descriptor with a new timestamp field that is initialized with the global value on the start of a transaction (`stmStart`) and updated each time the transaction validates its read set.

For cache-line based conflict detection, the STM runtime maintains a table of *transaction records*. Each record holds either a pointer to the descriptor of the transaction that currently owns the record (an exclusive lock for writing) or an odd-numbered value that is the timestamp of the last transaction that owned it. A hash function maps addresses to entries in this table. The hash function masks the lower 6 bits of an address (to provide

cache-line granularity) and a few of the upper bits (depending on the table size).

```

uint32
stmStart( TxnDesc* desc,
          TxnMemento* memento) {
    init(); /* init transaction structures */
    /* remember the current timestamp */
    desc->txTimeStamp = globalTimeStamp;
    return(desc->txTimeStamp);
}

uint32
stmRead(TxnDesc* desc, uint32* txnRec) {
    do {
        /* get the txnRec contents */
        v1 = *txnRec;
        if (v1 is a timestamp) {
            if (v1 > desc->txTimeStamp) {
                /* may abort directly */
                stmValidate(desc);
                /* update time */
                desc->txTimeStamp = v1;
            }
            log <v1, txnRec> in the read set;
            return v1; /* good to go */
        }
        if (v1 == desc) /* I own the lock */
            return(v1); /* good to go */
        /* handle contention */
        stmHandleContention();
    } while (1);
}

void
stmWrite(TxnDesc* desc, uint32* txnrec) {
    do {
        /* get the txnRec contents */
        v1 = *txnRec;
        if (v1 is a timestamp) {
            acquire ownership of the txnRec;
            if (v1 > desc->txTimeStamp){
                /* may aborts directly */
                stmValidate(desc);
                /* update time */
                desc->txTimeStamp = v1;
            }
            log <v1, txnRec> in write set;
            return;
        }
        if (v1 == desc) /* I own the lock */
            return;
        /* handle contention */
        stmHandleContention();
    } while (1);
}

void
stmCommit(TxnDesc* desc) {
    timeStamp = lockInc(globalTimeStamp);
    if (desc->txTimeStamp < timeStamp
        && write log is not empty)
        /* may aborts directly */
        stmValidate(desc);
    for <txnRec, v1> in write set
        /* release ownership */
        *txnRec = timeStamp;
    checkStableState(desc); /*wait if needed*/
}

void
stmCheckRead(TxnDesc* desc,
             uint32* txnRec, uint32 ver) {
    /* check version changed or locked */
    if (*txnRec != ver && *txnRec != desc)
        stmAbort(desc);
}

```

Figure 7. STM Algorithm

Figure 7 shows the STM algorithm. The `stmStart` function begins a transaction by initializing the transaction structures and storing the global time stamp in

the transaction descriptor. It returns the timestamp so that the compiler can inline some of the consistency checks. The `stmCommit` function validates the read set (rolling back if it fails), increments the global timestamp (using an interlocked increment instruction), and releases ownership of all its transaction records by storing an updated time stamp into the records; the transaction records, therefore, contain the timestamp of the latest transaction to have modified a datum. The `stmCommit` skips validation if the transaction is read only since by design the transaction would have aborted earlier if it had become inconsistent.

The `stmRead` and `stmWrite` barriers first check the transaction record of the accessed location to make sure no other transaction owns the record. The memory barriers also make sure that the read set of the current transaction is still valid if the timestamp of the accessed location indicates that the location's value comes from a transaction that committed after the current transaction started. If validation succeeds, we update the local timestamp which indicates the latest time (global timestamp) that the current transaction was valid. It is important to note that the STM performs a validation only when there is a potentially conflicting update from another transaction. Specifically, if two transactions T1 and T2 update disjoint data and T1 commits while T2 is executing, T1's commit will not cause a validation in T2. Since the STM updates locations in-place, we use the `stmCheckRead` to check that the transaction record hasn't changed after reading the datum to prevent reading speculative values (see Section 0). The `stmWrite` acquires ownership of the transaction record with the transaction subsequently doing an in-place update (after logging the old value with the `stmUndoLog` function). The interested reader may refer to [29] for details of the nesting, undo-logging and validation. We elide details of the contention management due to space constraints.

Aborting a transaction is more complex in an unmanaged language due to aliasing into stack allocated variables. Consider the example in Figure 8. In the function `bar` in Figure 8, the compiler will generate an undo log for the writes using the pointer `ptr`. These writes are into the stack; therefore, in general, the compiler generates undo logging for writes into the stack. Assume that the function `bar` updates several array elements using the pointer `ptr`. Suppose the transaction aborts immediately after the call to `foo` – the STM abort function will then use the same stack space as the temporaries in `foo` and the function `bar`. The STM abort function will undo all the writes, and undoing the writes in the function `bar` will overwrite (and corrupt) the abort routine's stack.

We avoid this problem by detecting when an undo may corrupt the stack. When a transaction starts, we need to take a snapshot of the register state (including the stack

pointer) so that the transaction can be re-executed on an abort. The stack space below the checkpointed stack pointer (assuming the stack grows downward) is dead after an abort. While undoing the writes, we check whether an address lies between the checkpointed stack pointer and the current stack pointer. If so, we avoid undoing the write to protect the stack.

Our runtime also supports a scalable transactional memory allocator via a customized malloc/free package [17]. It is aware of optimistic read concurrency so that it does not prematurely free memory.

```
#pragma tm_atomic
{
    foo();
    ... /* transaction aborted
        and calls the abort function */
}
int foo() {
int a[100];
...
bar(&a[0]);
}
...
int bar(int *ptr) {
*ptr = ... /* generates undo logging */
*(ptr + 4) = ...
}
```

Figure 8: Handling aborts

<pre>int Foo(int arg) { int tp; #pragma tm_atomic { b = a + 5; tp=a + 10; } ... }</pre>	<pre>int Foo(int arg) { int tp; jmpbuf env; TxnMemento* mem; TxnDesc* desc = stmGetTxnDesc(); checkpointLocalVars(); while(setjmp(&amp;env)) { recoverCheckpoint(); } stmStart(desc, mem); L1 = IRComputeTxnRec(a); V1 = IRRead(desc, L1); Temp1 = a; stmCheckRead(desc, L1, V1); IRWrite(desc, b); IRUndoLog(desc, b, ID); b = Temp1 + 5; L2 = IRComputeTxnRec(a); V2 = IRRead(desc, L2); Temp2 = a; stmCheckRead(desc, L2, V2); IRWrite(desc, tp); IRUndoLog(desc, tp, ID); tp = Temp2 + 10; stmCommit(desc); ... }</pre>
---	---

Figure 9: STM code generation

A TM system with optimistic readers can give incorrect results on programming idioms such as privatization. This arises because a doomed transaction may access (update or undo) data that it could not access if transactions truly executed in isolation, for example [17] shows how this can affect memory management.

To address these problems we use the mechanism of [17] to quiesce transactions. We define a transaction to

have reached a quiescent state if it commits, aborts and finishes undoing, or validates. At runtime, the STM maintains a linked list of active transactions [17]. A transaction inserts itself (adds its descriptor) to this list when it starts at the outermost level. Every time a transaction validates itself, it caches the current `globalTimeStamp` into the descriptor – i.e. the `txTimeStamp` field in the descriptor contains the latest timestamp at which the transaction was quiescent. After committing, a transaction T1 traverses the active transaction list. If there exists an active transaction T2 that started before T1 committed, and last quiesced before T1 committed, then T1 waits for T2 to quiesce (commit, abort, or validate) before unlinking from the list and proceeding. Note a long running transaction can not starve short running transactions since a transaction quiesces at validation, and validations happen at reads.

## 5. Compiler optimizations

This section describes the code generation and the optimizations for transactions. Our compiler performs redundant barrier elimination, inlines the fast paths of the STM routines, and optimizes the checkpointing of state at the start of a transaction.

The second column in Figure 9 shows the code generated from the transaction in the first column. We use `setjmp/longjmp` to implement the abort control flow. We also save the live temporaries into the transaction. The abort uses a `longjmp` to return back to the start of the transaction and to recover the temporaries.

```
int Foo(int arg) {
    int tp;
    jmpbuf env;
    TxnMemento* mem;
    TxnDesc* desc = stmGetTxnDesc();
    checkpointLocalVars();
    while(setjmp(&env)) {
        recoverCheckpoint();
    }
    stmStart(desc, mem);
    L1 = IRComputeTxnRec(a);
    V1 = IRRead(desc, L1);
    Temp1 = a;
    stmCheckRead(desc, L1, V1);
    IRWrite(desc, b);
    IRUndoLog(desc, b, ID);
    b = Temp1 + 5;
    Temp2 = a;
    stmCheckRead(desc, L1, V1);
    tp = Temp2 + 10;
    stmCommit(desc);
    ...
}
```

Figure 10. Optimized STM code

We introduce four new IR constructs (`IRComputeTxnRec`, `IRRead`, `IRWrite`, and `IRUndoLog`) representing the STM runtime functions. Since we need to ensure that the reads of a transaction are consistent throughout the transaction, we can not eliminate the check (`stmCheckRead`) that follows every transactional read, and hence we leave it as a function call

in the intermediate representation. For a non-transactional function containing a transaction, we load the transaction descriptor at the beginning of the function and store it into a temporary. The descriptor is passed as an implicit argument to the transactional version of a function. Since the loading of the transaction descriptor may be eliminated by common subexpression elimination, we leave it as a function call in the IR. Read and write barriers can be eliminated without regard to nesting, but undo logging can be eliminated only within the same nesting scope since we support partial rollback. Therefore the `IRUndoLog` construct takes an additional ID field that indicates the nesting depth. We then apply partial redundancy elimination (PRE) to the IR constructs to eliminate redundant STM operations. The PRE uses the nesting ID to avoid eliminating undo logging at different depths. We also filter out barriers to local variables – any temporary that does not have its address taken does not need a barrier. Our compiler also eliminates a read barrier that follows a write barrier to the same address since the write barrier already takes exclusive ownership of the transaction record. Applying these optimizations to the code in column 2 of Figure 9 results in the code in Figure 10. Our approach also ensures that redundant barriers are hoisted out of loops since loop invariant hoisting is a special case of partial redundancy elimination.

The live-in registers to a transaction must be saved (checkpointed) before entering the transaction. When the transaction rolls back, the saved registers must first be restored before the transaction can be re-executed.

During compiler optimizations, registerizable variables are identified as register candidates (or simply *registers*) in the intermediate representation. For the transaction shown in Figure 11 (a), it appears that only r2 should be checkpointed and the checkpointing code could be inserted as shown in Figure 11 (b). However, the compiler may optimize the code in Figure 11 (b) to that shown in Figure 11 (c), where r3 becomes live-in after it is moved above the transaction, and v4 becomes live-in after the assignment is moved into the transaction. If r3 and r4 are not checkpointed, the transaction roll-back may not roll back correctly.

In a highly optimizing compiler such as ours, it is difficult to restrict code motion of register candidates; for example memory fences would not restrict them in Figure 11 (d). One could try to determine live-in registers and perform the checkpoint after all the optimizations have been performed. However, this approach introduces two inefficiencies: 1) the checkpointing code to save and restore the live-in registers is not optimized, 2) some of the live-in values may have been spilled into memory and the inefficient memory logging method must be used to backup those spilled register variables.

<pre> r2 = r0 + 1 ... #tm_atomic {   r1 = r2 + 3   r3 = 99   ... = r3 } r5 = r4 </pre>	<pre> r2 = r0 + 1 ... mem2 = r2 #tm_atomic {   r2 = mem2   r1 = r2 + 3   r3 = 99   ... = r3 } r5 = r4 </pre>
<b>(a)</b>	<b>(b)</b>
<pre> r2 = r0 + 1 ... mem2 = r2 r3 = 99 #tm_atomic {   r2 = mem2   r1 = r2 + 3   ... = r3   r5 = r4 } </pre>	<pre> #tm_atomic {   m_fence();   r1 = r2 + 3   r3 = 99   ... = r3 } m_fence(); r5 = r4 </pre>
<b>(c)</b>	<b>(d)</b>

**Figure 11. Live-in registers to a transaction**

Our algorithm for register checkpointing is summarized in Figure 12. It augments the control flow graph for a transaction with a recovery block (recoverCheckpoint() in Figure 9) and allows traditional optimizations to be applied across transaction boundaries. It also tries to use available registers to store checkpointed values whenever possible, and only checkpoints those registers that are truly live-in to the transaction after all the compiler optimizations are performed.

Figure 13 shows the transformations for checkpointing a transaction without a function call (a) and with a function call (b). Any compiler optimization based on the program control and data flow information can be performed across transaction boundaries. For Figure 13(a), register allocation can not overwrite the live-in register value saved in  $t\_bkup\_n$  and  $t\_bkup\_r$  until the transaction successfully commits. Even when memory is used to save checkpointed values (Figure 13(b)), the compiler can still optimize the program across the transaction boundary. For example, the code in Figure 13 (b) can be optimized to Figure 13(c) with Partial Redundancy Elimination (PRE). After the optimization, the execution without transaction abort only needs to perform the computation  $n+1$  once instead of twice. Notice that, the code in Figure 13(b) and (c) uses memory locations to save the live-in registers and does not need the backward arcs to prevent the register allocation from clobbering the saved values.

```

Algorithm regCheckpoint
create a new recovery block
connect the new block to the entry block
live-in = {registers defined before and
used in/after the transaction}
for each r in live-in
  if no function call in the transaction
    t_bkup_r = a new virtual register
    insert "t_bkup_r = r" before the
    transaction entry
    insert "r = t_bkup_r"
    in the recovery block
  else /* there is a function call */
    m_bkup_r = a new stack location
    insert "m_bkup_r = r"
    before the transaction entry
    insert "r = m_bkup_r"
    in the recovery block

if no function call in the transaction
  add a fake control flow edge from the
  end of the region to recovery block

/* after compiler optimizations */
for each t in live-in
  if (t is unmodified inside transaction)
    remove the checkpointing code for t

```

**Figure 12. Register checkpointing algorithm**

## 6. Experimental results

The scheme described above has been implemented in Intel's `icc` compiler, for both C and C++ programs, running on either Windows or Linux operating system for IA32. We evaluate the implementation on a 16 processor SMP IBM x445 system [18] using 2.2GHz Xeon processors. The 16 processors share 16GB main memory and are grouped in to four clusters, with each cluster of 4 processors sharing a 64MB L4 cache. The programs are compiled with the optimization level `-O3`, without interprocedural optimizations (IPO). We compare the performance of transactional and lock-based code using the SPLASH-2 benchmarks [34] and a set of concurrent data structures. For the SPLASH-2 benchmarks, since we cannot support I/O inside transaction, we comment out the `printf` functions inside transaction (critical sections in lock version) since those were debugging aids – the useful I/O was outside critical regions which we handle normally. For the data structures, we use 80% lookups and 20% updates, with the data pre-populated before the runs.

Figure 14 shows the effect of compiler optimizations on the single-thread performance for three data structures. The Y-axis shows execution time normalized to the locking version (**fine-lock**). On the average, STM version (**compiler stm**) performs within 63% overhead over the locking version. Compared to the code without optimization (**no opt**), the STM performance benefits from the compiler optimizations. Inlining the STM fast paths provides the most benefit followed by elimination of redundant barriers and register checkpointing optimizations.

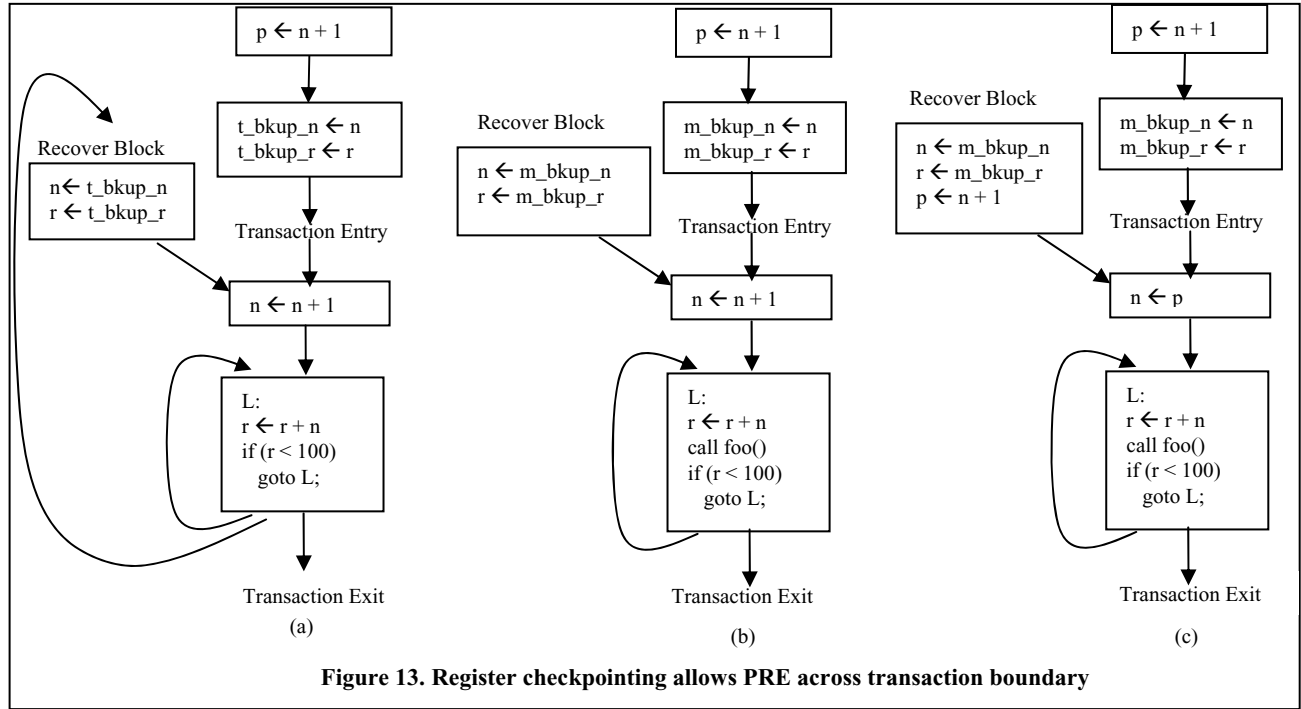


Figure 13. Register checkpointing allows PRE across transaction boundary

Figure 15 further compares STM version with the manually transformed code for STM (the bars marked **manual**). The manual version is about 26% slower than the locking version. However, the manual version takes advantage of the fact that strcmp/strepy routines can be called inside Hashtable and Avltree without instrumentations, although the routines have to be instrumented for Btree to work correctly. If the user can tell the compiler to instrument strcmp/strepy for Btree but not for Hashtable and Avltree, the STM version (**compiler stm + str opt**) is only about 21% slower than the locking version and performs slightly better than the manual version since some compiler optimizations are not applied in hand-coded version.

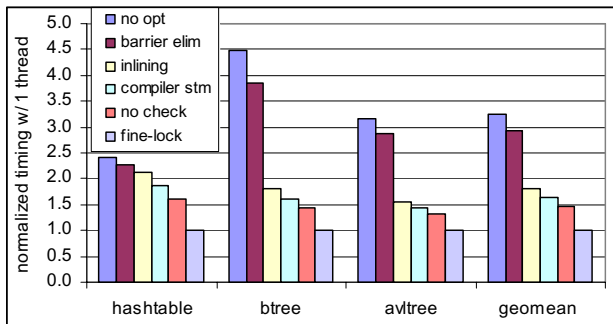


Figure 14. Optimization benefits for structures

Figure 16 shows the benefit of compiler optimizations for five SPLASH-2 benchmarks executed with single-thread. The remaining SPLASH-2 benchmarks spend

little time inside critical sections. The STM overhead ranges from 0% for Cholesky to 15% for Barnes, with an average of 6.4%. The overhead is smaller since the application spends a smaller proportion of time inside transactions.

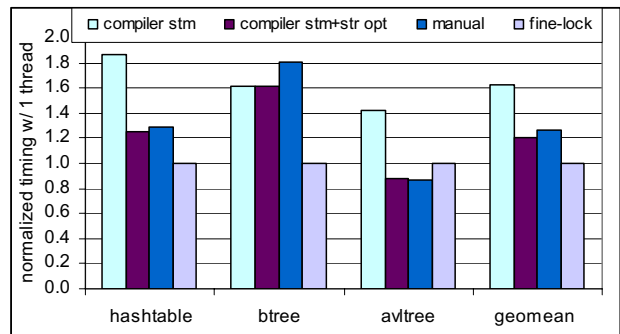


Figure 15. Comparison with manual optimizations

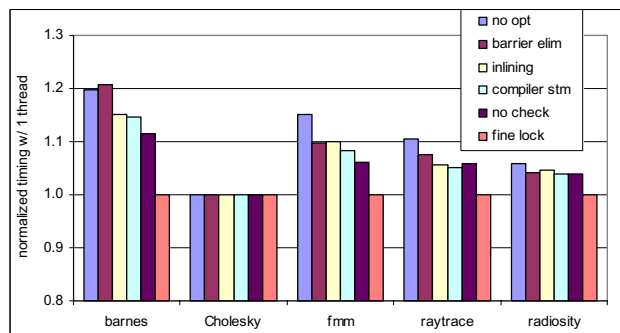


Figure 16. Optimization benefits for SPLASH-2

Figure 17 shows the performance on 16 threads (on 16 CPUs) for the three data structures, normalized to the fine-locking version. The coarse lock versions use the same global lock for all the critical sections in the program, while the fine lock versions may use different locks for different critical sections. The STM versions of the benchmarks (compiler stm) perform in most cases better than the lock-based version. Figure 18 shows the 16-thread performance for SPLASH-2. The STM versions perform comparably to fine-grain locking code, while STM provides the programming ease of coarse-grain locks.

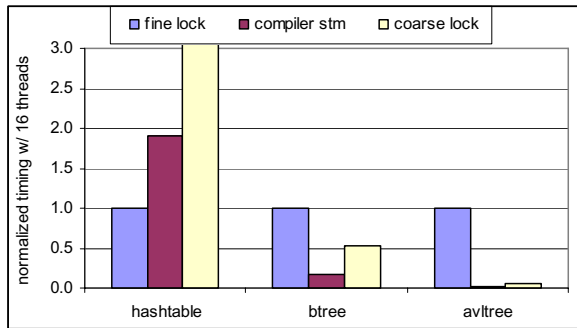


Figure 17. 16 threads performance for structures

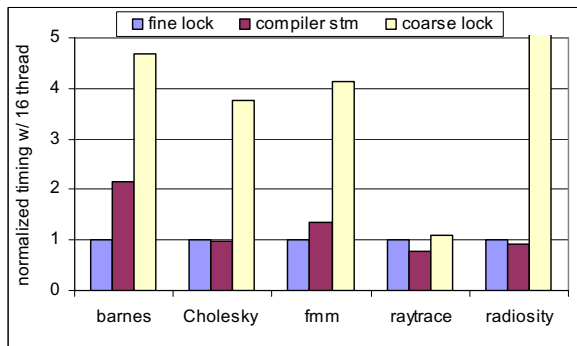


Figure 18. 16 threads performance for SPLASH-2

Figure 19 and Figure 20 show the scalability of the Barnes and Raytrace benchmarks. For Raytrace, the STM version scales better than the fine-grain version, but for Barnes the STM version does not scale as well as the fine-grain version, due to its large read set. Note that the transactional version always performs better than coarse-grain locking even though the transactional version uses similar coarse-grain concurrency controls.

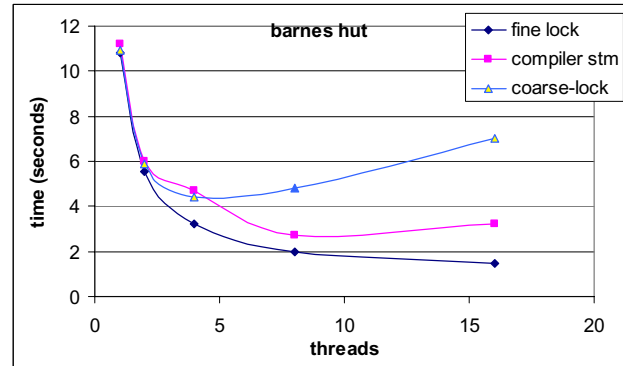


Figure 19. Scalability of Barnes Benchmark

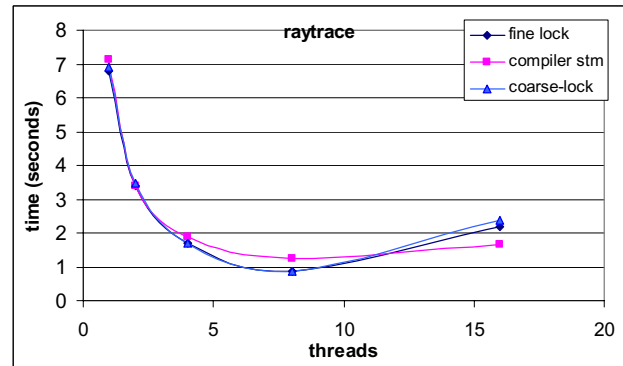


Figure 20. Scalability of Raytrace Benchmark

Figure 21 shows the characteristics for the SPLASH-2 benchmarks. Five of the SPLASH-2 benchmarks have reasonable amount of execution time in the critical sections, while the last three have little execution time inside critical sections (average < 1%). The average size of read set is relatively small, up to 34 words (4 bytes each word) per transaction. For the five benchmarks, the number of validations ranges from 57.5K to 1.2M, and the number of aborts reaches about 67.8K for Radiosity. Interestingly, Fmm needs fewer validations than the total number of transactions because our time-stamp based validation scheme can avoid unnecessary validations on some read-only transactions. From this table, we can see that the good performance of Raytrace is due to its smaller read set and fewer validations, and the poor performance of Barnes comes from its larger read set.

Name	%time In CS	Rset size	#txn 16T	#val 16T	#abt 16T
Radiosity	22.0	3.2	916K	1.2M	68K
Cholesky	10.5	5.8	41K	58K	3K
Raytrace	9.0	2.8	153K	154K	49K
Barnes	8.0	15.4	276K	283K	2k
Fmm	5.8	34.2	167K	143K	161
Water-Ns	2.3	31.9	48K	48K	137
Ocean	0.1	2.0	3K	2592	0
Water-Sp	0.1	1.3	305	481	0

Figure 21. Characteristics of SPLASH-2 benchmarks

Figure 22 shows code size expansion from STM compilation and optimizations. The lock version of the programs was subject to the same degree of optimization as the STM versions. The basic transformation without fast-path inlining increases the code size by about 4% over the fine lock version for the five SPLASH-2 benchmarks. Together with fast-path inlining, STM compiler generated code is about 14% bigger than the lock version. The compilation time increase is negligible, as the STM optimizations are light-weight compared to the other compiler analysis and optimizations.

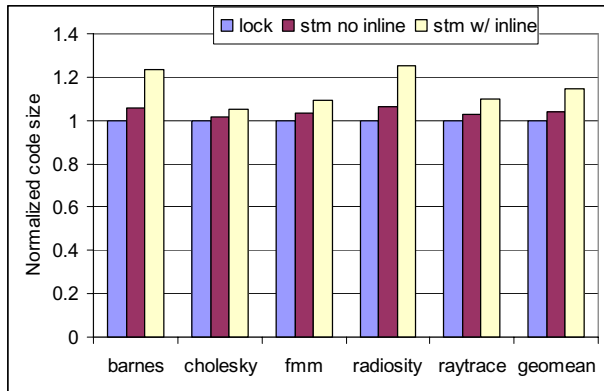


Figure 22. Code size expansion with STM compilation

## 7. Related work

The closest prior work is the research on adding transactional memory to managed languages [1][4][11][13][10][28]. Adding transactions to unmanaged languages poses additional challenges due to the lack of type-safety, exception handling, and local variable aliasing. Our user-level abort primitive is similar to the retry construct in [13]. Nested transactions (both closed and open) were discussed in [16]. The HPCS languages [2][5][6] have also defined language level transaction constructs in lieu of locks.

Software transactional memory (STM) was first introduced in [31], where the user had to statically declare the set of memory locations touched by the transaction. Subsequently, [11][15] came up with STM implementations where the set of memory locations did not need to be specified upfront. These initial STM implementations used non-blocking algorithms and incurred a very high overhead. Practical high performance STM algorithms that relied on 2-phase locking were first described in [29]; these algorithms were subsequently used in the managed language implementations [1][10]. TARIFA [33] also supports *transaction* in C/C++ programs. However, it does not work with a compiler. Instead, the transaction is supported via assembly code instrumentation, which incurs heavy runtime overhead. Furthermore, it does not deal with features such as function pointers.

A time-stamp based approach is proposed by Riegel et al. [7], which maintains list of ranges to ensure consistency in a non-blocking STM runtime. Our approach maintains a single consistency point so is more efficient. Dice et al designed a TL2 algorithm with a global-clock to achieve consistency [27]. TL2 aborts a transaction as soon as a datum is found later than the local timestamp, while we perform a read-set validation in that situation and may avoid the abort. TL2 employs write buffering and commit-time locking, while ours performs in-place update and encounter-time locking. Finally, TL2 does not specify how to address privatization issues.

Other researchers have also investigated implementing transactional memory in hardware, but none of these have addressed language level extensions. Transactional memory was first proposed in [14], which used cache coherency to support bounded transactions. Subsequently, other researchers [3][9][26][22][24] have discussed how virtualized transactions may be implemented in hardware. Hardware implementations of transactional memory provide a performance advantage, but come at a significant hardware complexity. Moreover, hardware implementations of transactional memory often come with semantic limitations – for example, lack of condition synchronization, abort on context switches, etc. Hybrid transactional memory [19][23][32] proposes to combine hardware and software transactional memory implementations. Transactions are first tried in HW, but if they can not meet the hardware constraints (such as size, duration, and semantics), then they are re-executed as a software transaction.

## 8. Conclusions and future work

In this paper, we describe novel code generation and optimization techniques for transactional memory constructs in C/C++, and address a number of challenging issues that do not arise in managed environments. We have implemented these mechanisms in a production-quality C compiler and a high-performance software transactional memory runtime. We measure the effectiveness of these optimizations and compare the performance of lock-based versus transaction-based programming on the well-known SPLASH-2 benchmark suite. Our results show that in an unmanaged environment, transactions can perform comparable to fine-grain locking, but provide the programming ease of coarse-grain locking. When running with 16-processors, the transactional version performs better than the fine-grain version in most cases. On average, the compiler generated code is about 6.4% slower than the lock-based version on a single thread for SPLASH-2 benchmarks.

There are a number of open issues that we want to address in future work. We currently don't support I/O or system calls inside transactions. We also don't consider threading or signal handling inside a transaction.

## Acknowledgements

We would like to thank Intel's compiler team for developing the compiler infrastructure used in this study, Jesse Fang for his support, Shiliang Hu and Wei Liu for their valuable comments. We appreciate the comments from the anonymous reviewers that helped improve the quality of the paper.

## References

- [1] Adl-Tabatabai, A., Lewis, B.T., Menon, V.S., Murphy, B.M., Saha, B., Shpeisman, T. Compiler and runtime support for efficient software transactional memory. *PLDI 2006*.
- [2] Allen, E., Chase, D., Luchango, V., Maessen, J., Ryu, S., Steele Jr., G., Tobin-Hochstadt, S. The Fortress language specification, version 0.618. Sun Microsystems Technical Report, April 2005.
- [3] Ananian, C.S., Asanovic, K., Kuszmaul, B.C., Leiserson, C.E., Lie, S. Unbounded Transactional Memory. *HPCA 2005*.
- [4] Carlstrom, B., Chung, J., McDonald, A., Chafi, H., Kozyrakis, C., and Olukotun, K. The Atomos transactional programming language. *PLDI 2006*.
- [5] Charles, P., Donawa, C., Ebcioğlu, K., Grothoff, C., Kielstra, A., von Praun, C., Saraswat, V., Sarkar, V. X10: An object oriented approach to non-uniform cluster computing. *OOPSLA*, October 2005.
- [6] Cray Inc. The Chapel language specification, version 0.4. Technical Report, Cray Inc. Feb 2005.
- [7] Dice, Dave, Ori Shalev, Nir Shavit, Transactional Locking II, 20th International Symposium on Distributed Computing (DISC 06), Stockholm, Sweden, September 18 - 20, 2006.
- [8] Geppert, L. Sun's Big Splash, IEEE Spectrum, January 2005
- [9] Hammond, L., Carlstrom, B.D., Wong, V., Hertzberg, B., Chen, M., Kozyrakis, C., and Olukotun, K. Transactional coherence and consistency. *ASPLOS 2004*.
- [10] Harris, T., Plesko, M., Shinnar, A., and Tarditi, D. Optimizing Memory Transactions. *PLDI 2006*.
- [11] Harris, T.L. and Fraser, K. Language support for lightweight transactions. *OOPSLA 2003*.
- [12] Harris, T.L. Design choices for language based transactions. University of Cambridge Computer Laboratory, Tech Report, Aug 2003.
- [13] Harris, T.L., Marlow, S., Peyton Jones, S., Herlihy, M. Composable memory transactions. *PPoPP 2005*.
- [14] Herlihy, M. and Moss, J.E.B. Transactional memory: architectural support for lock-free data structures. *ISCA 1993*
- [15] Herlihy, M., Luchango, V., Moir, M., Scherer III, W.M. Software transactional memory for dynamic sized data structures. *PODC 2003*.
- [16] Hosking, A, Moss, J.E.B. Nested transactional memory: Model and preliminary Sketches. *SCOOOL 2005*.
- [17] Hudson, Richard L., Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg, McRT-Malloc - A Scalable Transactional Memory Allocator, ISMM '06.
- [18] IBM Corp., Intel processor-based servers: x445, <http://www-03.ibm.com/servers/eserver/xseries/x445.html>.
- [19] Kumar, S., Chu, M., Hughes, C., Kundu, P., Nguyen, A. Hybrid transactional memory. *PPoPP 2006*.
- [20] Marathe, V. J., Scherer, W. N., and Scott, M. L. Adaptive software transactional memory. Technical report 868. Computer Science Department, University of Rochester, 2005.
- [21] Marathe, V. J., Scherer, W. N., and Scott, M. L. Design tradeoffs in modern software transactional memory systems. In *7th Workshop on Languages, Compilers, and Run-Time Support For Scalable Systems* (Houston, Texas, October 22 - 23, 2004).
- [22] McDonald, A., Kozyrakis, C., Olukotun, K. Architectural Semantics for Practical Transactional Memory, *ISCA 2006*.
- [23] Moir, M. Hybrid Transactional Memory. Sun Microsystems Technical Report.
- [24] Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A. LogTM: Log-based Transactional Memory. *HPCA-12*, 2006.
- [25] Rajwar, R., Goodman, J. R. Transactional lock-free execution of lock-based programs. *ASPLOS 2002*.
- [26] Rajwar, R., Herlihy, M., and Lai, K. Virtualizing transactional memory. *ISCA 2005*.
- [27] Riegel, T., P. Felber, and C. Fetzer, A Lazy Snapshot Algorithm with Eager Validation, 20th International Symposium on Distributed Computing (DISC 06), Stockholm, Sweden, September 18 - 20, 2006.
- [28] Ringenberg, M.F. and Grossman, D. AtomCaml: First-class atomicity via rollback. *ICFP 2005*.
- [29] Saha, B., Adl-Tabatabai, A., Hudson, R., Cao Minh, C., Hertzberg, B. McRT-STM: A high performance software transactional memory system for a multi-core runtime. *PPoPP 2006*.
- [30] Schouten, D, X. Tian, A. Bik, M. Girkar, Inside the Intel Compiler, Linux Journal, Volume 2003, Issue 106 (February 2003).
- [31] Shavit, N., and Touitou, D. Software transactional memory. *PODC 2005*.
- [32] Shriraman, A., Marathe, V.J., Dwarkadas, S., Scott, M.L., Eisnstat, D., Heriot, C., Scherer III, W.N., Spear, M.F. Hardware acceleration of software transactional memory. Technical report 887, Computer Science Department, University of Rochester, 2006.
- [33] TARIFA, <http://www.hackshack.de/>.
- [34] Woo, S. C., M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, The SPLASH-2 Programs: Characterization and Methodological Considerations, In Proc. of the 22nd Annual Intl. Symp. On Computer Architecture, Jun. 1995.
- [35] Ying, V., C. Wang, Y. Wu, X. Jiang, Dynamic Binary Translation and Optimization of Legacy Library Code in a STM Compilation Environment, *WBLA06* in conjunction with *ASPLOS-2006*.