Programming Support for Speculative Execution with Software Transactional Memory

Min Feng Rajiv Gupta Iulian Neamtiu
University of California, Riverside
Email: {mfeng,gupta,neamtiu}@cs.ucr.edu

***Abstract*—In this paper, we identify the practical issues that deter the adoption of software transactional memory (STM) for speculation in real applications. These issues include dealing with excessive instrumentation added by naive identification of potential shared accesses, functions that may be called from both transactional and non-transactional contexts, and calls to functions for which the source is unavailable. We address these challenges and provide an approach for developing speculatively-executed code in C/C++ that offers superior programmability and performance. Our contributions consist of a set of programming constructs for writing speculatively-executed code and a compiler that translates code annotated with these constructs into speculatively-executable code that uses STM runtime libraries. Our approach uses annotations that simply mark the boundaries of the code that is to be executed speculatively and supports calling precompiled (e.g., C standard library) and irreversible (e.g., I/O operations and system calls) functions from within transactions. We employ a series of important optimizations for reducing the overhead of speculative execution, including: placement of read/write barriers only for accesses that actually can cause a data race; elimination of redundant read/write barriers by caching shared variables; and eliminating unnecessary search in the write buffer.**

We evaluate the programmability of our constructs and the performance of our compiler implementation using eight STAMP benchmarks and two real applications—the Velvet genomic assembler and the ITI decision tree constructor. Compared to Intel's STM compiler, our approach requires 91% **fewer constructs to be inserted by the programmer, yet it achieves** 20.8% **better performance.**

I. Introduction

Speculative parallelization [1] has been proposed for executing interdependent code in parallel, while ensuring its original semantics. For example, speculative parallelization is often used to parallelize loops where cross-iteration dependences arise seldom at runtime—the loop executes in parallel by speculating on the absence of cross-iteration dependences. Each iteration is executed in a speculative state. To enforce correctness, the iterations are committed according to their order in the original code.

Programming speculative parallel code in an unmanaged language, such as C/C++, is a demanding task for programmers. A major part of this effort involves writing speculatively-executed code. Software transactional memory (STM) [2] has been widely used for speculative execution. So far researchers have not paid enough attention to the programming effort with STM. Because most STM implementations [3], [4] are in the form of libraries, programmers need to manually insert low-level STM API calls (e.g., read/write barriers) into the parallel code. Table I (column 2) summarizes the programming burden of writing speculatively-executed code using low-level STM API calls. First, programmers need to insert read/write barriers for each read/write that may access shared memory. In real applications, a transaction may contain hundreds of lines of code, and tens or hundreds of functions may be called directly and indirectly from within the transaction. *There may be hundreds of shared read/write barriers to be added, which is a great burden on programmers*. Second, some STM libraries [5] also require programmers to add STM-related arguments into the declaration of functions called directly and indirectly in a transaction. In real applications, *there may be hundreds of function declarations that need to be modified*. Finally, in real applications, the source code of functions such as precompiled library functions and system

	STM libs	Intel STM	Ours
Instrumenting shared access	Manual	Auto	Auto
Annotating speculative func.	Manual	Manual	Auto
Handling precompiled func.	Manual	Manual	Auto
Addressing system calls	Manual	Manual	Auto

Table I
PROGRAMMING BURDEN OF USING DIFFERENT STM IMPLEMENTATIONS. *Manual* — TASKS DONE MANUALLY BY THE PROGRAMMER; *Auto* — TASKS DONE AUTOMATICALLY BY THE COMPILER.

Program	Low-level STM API		Intel STM Comp.		Ours
	NOC	*XLOC*	*NOC*	*NOF*	*NOC*
Bayes	176	4,291	75	60	15
Labyrinth	98	1,839	31	28	3
Genome	122	477	31	26	5
Intruder	265	4,408	95	92	3
Kmeans	17	73	3	0	3
Ssca2	55	2,065	10	0	10
Vacation	359	4,241	151	148	3
Yada	324	4,710	115	109	6
Average	*177*	*2,750*	*64*	*58*	*6*

Table II
PROGRAMMING EFFORT FOR WRITING SPECULATIVELY-EXECUTED CODE IN C/C++. NUMBERS WERE MEASURED USING THE STAMP BENCHMARK SUITE. *NOC*—NUMBER OF PROGRAMMING CONSTRUCTS INSERTED FOR SPECULATIVE EXECUTION; *XLOC*—SOURCE LINES OF CODE THAT NEED TO BE EXAMINED; *NOF*—NUMBER OF FUNCTIONS CALLED DIRECTLY AND INDIRECTLY IN THE TRANSACTION.

calls may not be available to the programmer. The low-level STM API lacks support for these functions, which deters using STM for speculation in real applications.

To illustrate the programming burden of using STM and call attention to this problem, in Table II we show the programming effort for writing speculatively-executed code in C/C++ using low-level STM API constructs (columns 2–3). We collected the data from the STAMP benchmarks, which contain hand-coded transactions using low-level STM APIs. The reported numbers include the number of programming constructs inserted in the library functions called in these benchmarks. To use STM in these benchmarks, the programmer needs to insert on average 177 programming constructs for each application. Each application has on average 58 functions called directly and indirectly in transactions. Programmers need to examine the source code in these functions line by line to find all shared reads/writes. The table shows that on average 2,750 lines of code need to be examined for each application. Moreover, the code to be examined is usually spread across multiple files, which further increases the programming effort.

The Intel C++ STM Prototype Compiler [6], [7], [8] and OpenTM [9] provide first-class STM constructs for C/C++. However, these approaches still present challenges to programmers, which are summarized in Table I (column 3). First, they still require programmers to annotate the declaration of each function called directly and indirectly from within a transaction. Therefore, when many functions are called in the transaction, these programming constructs impose a significant burden on the programmer. Table II shows the programming effort imposed by the Intel STM compiler (columns 4–5). For each STAMP benchmark, programmers need to insert on average 64 programming constructs, around 91% of which are used for annotating function declarations; moreover, this effort must be repeated for each new program that is converted to use speculative execution. Second, the two compilers cannot transact-ify precompiled library functions whose source code is unavailable. Therefore, if a transaction calls any precompiled library function in transactions, the compilers simply serialize the execution of that transaction, i.e., no other transactions are allowed to execute in parallel with the serialized transaction [8]. As a result, Intel STM and OpenTM cannot be used to speculatively parallelize code containing precompiled library functions, a situation commonly encountered in real applications. In contrast, we were able to parallelize two real applications using the approach presented in this paper—Velvet, a widely-used genomic assembler [10], and Incremental Tree Inducer (ITI), a well-known decision tree constructor [11].

In this paper, we identify and address the challenges that deterred the adoption of STM for speculation in real applications. Our solution is a combination of annotations and analyses. We introduce a set of programming constructs to writing speculatively-executed code in C/C++. Unlike previous STM approaches [6], [9], our constructs do not require programmers to annotate each function called in a transaction. Rather, programmers just need to specify transaction boundaries. We also provide support for precompiled library functions (e.g., C standard library) whose source code is unavailable. Programmers can annotate the library functions using our constructs to enable concurrent execution of the transactions that contain these functions. For system calls whose results cannot be rolled back (e.g., I/O operations and system calls), we provide constructs to suspend the calls to these library functions inside transactions and execute them outside. The last column in Table II presents the annotation effort of our approach—note the order-of-magnitude reduction compared to the other two approaches.

We present the design of our compiler that transforms annotated C/C++ code into speculatively-executable code using write buffering-based STMs. The programmers do not need to annotate the functions called in transactions as our compiler automatically classifies all functions in a program into four groups based on their usage in transactions. We apply different code transformations to different groups for performance and correctness. Our compiler automatically inserts low-level STM API calls (e.g., read/write barriers) into transactions. To optimize transaction performance, we use static data race analysis to avoid placing read/write barriers for reads/writes of global variables that do not cause data races. We cache the values loaded/stored by shared reads/writes to eliminate redundant read/write barriers. Since the overhead of write buffering-based STMs comes largely from searching write buffers [12], our compiler eliminates unnecessary searches in write buffers for data that is definitely not present in write buffers.

Our compiler is implemented as a source-to-source translator based on the ROSE [13] program transformation infrastructure. We evaluate our implementation on a Dell PowerEdge T605 server (Intel Xeon 8-core processor) using the STAMP benchmark suite and two additional real applications—Velvet and ITI. We parallelized all programs using speculative parallelism. Compared to the Intel STM compiler, our approach requires 91% fewer programming constructs to be inserted into the STAMP benchmarks. On average, our compiler implementation achieves 1.62x speedup for the ten applications, using 8 threads. Compared to the Intel STM compiler, our approach improves the performance of the STAMP benchmarks by 20.8%.

The rest of the paper is organized as follows. Section 2 presents our programming and compiler support for writing speculatively-executed code. Section 3 describes our support for enabling calling of precompiled library functions and irreversible functions within transactions. Section 4 presents a set of optimizations to reduce runtime overhead. Section 5 evaluates our implementation. Section 6 discusses related

work and Section 7 concludes this paper.

II. Speculatively-executed Code

This section presents our programming constructs for writing speculatively-executed code and describe the code translation for programs annotated with these constructs.

A. Programming Constructs

To facilitate speculative parallelization, we only require programmers to specify the boundaries of a transaction:

```
#pragma tm transaction
{
  ... // statements here
}
```

Unlike traditional STM libraries [3], [4] or the Intel STM compiler [6], our constructs do not require programmers to manually insert read/write barriers into transactional code or annotate functions called within transactions. Transactions can include function calls. The compiler instruments shared reads/writes within transactions so that the STM runtime system can detect conflicting accesses to shared variables to guarantee atomicity and isolation. Upon a conflict, the STM runtime rolls back execution and re-executes the transaction.

B. Code Compilation

Unlike most previous STM compilers [14], [15] that target managed languages such as JAVA, our programming constructs are designed for C and C++, which are unmanaged languages. Compared to managed languages, supporting our programming constructs in C/C++ is more challenging. First, for C/C++ programs, our compiler must generate code statically—we cannot use JIT compilation as managed languages do. JIT compilers can perform operations such as creating an atomic clone for a function on-the-fly, dynamically suppressing redundant/dead barriers, and transactifying library functions by inlining them, which static compilation cannot do. Second, due to the unsafe use of pointers in C/C++ programs, our compiler is forced to use word-based STMs while the compilers for managed languages can use object-based STMs. Finally, without type safety, shared reads/writes should be checked conservatively, which further makes reducing STM overhead quite challenging.

Our compiler translates programs annotated with our programming constructs into source code instrumented with calls to an STM runtime library. Figure 1 shows the process of code generation, where the grey blocks indicate the work done by our compiler. The user code is written in C/C++ with our programming constructs. Our compiler first takes the user code as input and translates the functions and calls based on the call graph. It then instruments code with low-level STM API constructs and uses static data race analysis to find shared data accesses, as described in detail in Section II-B2. Finally, the compiler eliminates redundant read and write checks to reduce the overhead, which will be described in Section IV. The generated code is C/C++ code with calls to low-level STM functions, and can be compiled with regular compilers, such as GCC, to generate an executable binary.

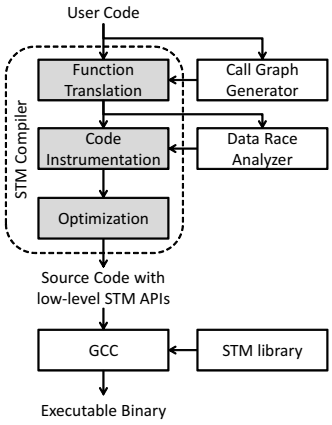

Figure 1. Overview of code generation.

1) Function Translation: This first step in code translation is to translate all functions defined in the user code according to their call sites and their usage via function pointers. To do this, our compiler classifies the functions into the four following types, using the call graph generated by static analysis.

i. Atomic functions are functions called only inside transactions. In other words, their call sites are either in transactions or in other atomic functions. These functions need to be executed atomically. Therefore, the compiler must instrument the shared reads/writes in them with low-level STM API constructs.

ii. Non-atomic functions are functions that are never called in any transaction, i.e., these functions are called either outside transactions or from non-atomic functions. There is no need to do any special code translation for them.

iii. Double-duty functions are functions called both inside and outside transactions. Our compiler creates atomic clones for such functions and instruments the clones with low-level STM API constructs. All calls to these functions that appear in transactions are replaced with calls to their atomic clones.

iv. Dynamically-called functions are functions called through function pointers. Since these functions may be called both inside and outside transactions, we must decide at runtime whether to call the original function or its atomic clone. To solve this problem, for each dynamically-called function, the compiler creates an atomic clone and places a conditional call to the atomic clone at the beginning of the original function. The code below gives an example: the original function checks a thread-local variable, `inside_transaction` at the beginning. The variable is set to be true when the current thread enters a transaction. If the variable is true, the original function then calls its atomic clone; otherwise, the original statements inside the function are executed.

```
int original_func()
{
  if ( inside_transaction == true )
    return atomic_func();
  // original statements here
}
```

API function	Description
txDesc* stmGetDesc()	Get a descriptor
void stmBegin(txDesc*)	Start a transaction
void stmEnd(txDesc*)	Validate & commit
void stmAbort(txDesc*)	Explicitly abort
Type stmRead⟨Type⟩(txDesc*, Type*)	STM read barrier
void stmWrite⟨Type⟩(txDesc*, Type*, Type)	STM write barrier
void stmReadBytes(txDesc*, void*, void*, size_t))	STM read barrier
void stmWriteBytes(txDesc*, void*, void*, size_t))	STM write barrier
void stmLogStack(txDesc*)	Log stack variables
void stmLogBytes(txDesc*, void *, size_t)	Log specific var.
(void*) stmMalloc(txDesc*, void*)	STM malloc
void stmFree(txDesc*, void*)	STM free

Table III
THE LOW-LEVEL STM API USED IN TRANSLATED CODE.

2) Code Instrumentation: After function translation, the compiler instruments the transaction code, atomic functions, and atomic clones of functions with low-level STM API constructs. Table III presents the low-level STM API calls used by our compiler. This API is designed for word-based STM libraries, such as TL2 [3] and TinySTM [4]. We chose word-based STM libraries rather than object-based libraries due to the lack of type safety and the presence of unsafe pointer arithmetic in C/C++ [7]. Although the compiler is designed to generate code for STM libraries, it can also be used for hardware and hybrid TM libraries as long as their implementations are compatible with the listed API.

All low-level STM functions require a transaction descriptor as an input. The transaction descriptor is obtained by calling the function `stmGetDesc`. Each thread has a unique transaction descriptor, which is created in thread-local storage when function `stmGetDesc` is called for the first time in the thread. Calls to function `stmGetDesc` are inserted before every transaction. To eliminate redundant accesses to thread-local storage within a transaction, a local variable is used to hold the transaction descriptor and passed to every called atomic function through arguments.

The compiler inserts the `stmBegin` and `stmEnd` API calls at the boundaries of transactions to start and commit them. The `stmAbort` call is inserted where explicit transaction abort is specified. The aforementioned three functions dynamically decide if the transaction is nested in another transaction by checking a thread-local variable `inside_transaction`. If so, they start/end the transaction as an inner transaction; otherwise, the transaction is treated as an outermost transaction. In previous works [7], [8], transactions are statically classified as outermost transactions and inner transactions and different API constructs are inserted to start/commit them. We use dynamic checks because a function containing transactions may be called from both inside and outside of other transactions. Therefore, the transactions in the function may be either outermost or inner transactions, depending on the function's call site.

Barrier functions perform the required STM operations to ensure consistency and detect conflicts for shared memory accesses in transactions. We have read/write barriers for each basic data type in C/C++. For user-defined data constructs, we use `stmReadBytes` and `stmWriteBytes` as read/write barriers. Our compiler instruments atomic code with read/write barriers as follows:

i. Find Shared Variables. It is very important for the compiler to only place read/write barriers at necessary places since they are usually time-consuming at runtime. Instrumenting all accesses to global/heap variables with read/write barriers usually introduces unnecessary barriers since some of global/heap variables may be read-only in transactions, or may not be shared across transactions. To avoid placing unnecessary read/write barriers, the compiler uses static data race analysis [16] to find potentially shared variables, i.e., variables that two transactions may access without synchronization and one of the accesses is a write.

ii. Normalize Operators. The compiler then normalizes C/C++ operators on these variables. In C/C++, a variable reference may be both a read and a write at the same time. Since our compiler works on C/C++ source code, it cannot directly insert read/write barriers for such variable references. For example, the reference to `a` in `a++` is both a read and a write. To enable barrier insertion, our compiler converts `a++` to `a=a+1`, where the first reference to `a` is a write and the second is a read.

iii. Insert Barriers. Finally, our compiler inserts read/write barriers for accesses to the potentially shared variables found in step 1. For example, after instrumentation, `a=a+1` will be `stmWriteInt(tx, &a, stmReadInt(tx, a)+1)`, where `tx` is the transaction descriptor.

Although we do not need to detect conflicts for writes to live-in private variables (including local variables on the stack, thread-local variables, and global variables that are not shared), the original values in these variables need to be logged to allow rollback if needed. The compiler inserts `stmLogStack` before each transaction to save the local state. Function `stmLogStack` uses the EBP and ESP registers on the x86 architecture to locate and save the local variables on the stack. Similar registers can also be found on other architectures, e.g., SP and LR on the ARM processors. The compiler also inserts `stmLogBytes` before every write to thread-local variables and global variables that are not shared. The function saves the value at an address if the address has not been logged in the transaction.

Functions `stmMalloc` and `stmFree` are the STM versions of `malloc` and `free`. Our compiler replaces `malloc` and `free` in transactions with the STM versions.

III. SUPPORT FOR LIBRARY FUNCTIONS

In previous works [7], [8], [9], two types of functions cannot be transact-ified: precompiled library functions and functions that cannot be rolled back (e.g., system calls and I/O operations). Calls to these types of functions within transactions can be detected by either the compiler or the runtime system. If a transaction is identified to contain such function calls, it is executed in serial mode, i.e., other transactions cannot run concurrently with it. This

makes it impossible to speculatively parallelize code that contains calls to precompiled or irreversible functions. In this section, we introduce two programming constructs to avoid serialization of transactions when such functions are called in transactions.

A. Precompiled Library Functions

We introduce the construct `#pragma tm precompiled` to annotate precompiled library functions so that the compiler can transact-ify them. The syntax is:

```
#pragma tm precompiled [read(...)] [write(...)]
// function declaration here
```

The construct is introduced immediately preceding a function declaration, and tells the compiler what memory locations the function reads and writes. A memory location can be a variable or a (pointer,size) pair. After a precompiled function is annotated with the construct, it can be used as a regular function in transactions. In a transaction where the function is called, the compiler creates local copies of the shared memory locations that the function accesses. The function then works on the local copies instead of directly accessing the shared memory locations. In this way, we eliminate the need to transact-ify the code inside the function since it does not touch shared memory locations. The compiler only needs to add read/write barriers when copying data between shared memory locations and local copies. The transactions with annotated library functions can thus be executed in parallel with other transactions. The code below provides an example with the `precompiled` construct used to annotate two library functions.

```
#pragma tm precompiled read(x)
float sin (float x);
...
#pragma tm precompiled \
     read(dst, src, num, (*src, num)) \
     write((*dst, num))
char * memcpy(void *dst, void *src, size_t num);
```

The first function is a mathematical function provided by the C mathematical library. We put argument `x` in the `read` clause since it is read in the function. The `write` clause is omitted since no variable is written by the function. The second function is a memory copy function provided by C string library. It copies the values of `num` bytes from the memory location pointed to by `src` to the one pointed to by `dst`. Since the function reads the `num`-byte memory block pointed to by `src`, we put `(*src, num)` in the `read` clause. Similarly, we put `(*dst, num)` in the `write` clause.

In certain cases it is impossible to know the memory locations accessed by a function until it is called. In such cases, we use the `precompiled` construct to annotate the call site of the function. In the example that follows, the function copies a string pointed to by `src` into the array pointed to by `dst`. The string ends in a null character. We cannot know the memory size accessed by the function at its declaration, since the string length is not fixed. However, we can conservatively annotate the function at its call site since the maximum length of the string (i.e., the size of the allocated memory block) is known at that time.

```
char src[256], dst[256];
#pragma tm transaction
{
  ... // statements here
  #pragma tm precompiled \
       read(dst, (*src, 256), src) \
       write((*dst, 256))
  strcpy(dst, src);
}
```

Calls to an annotated library function need to be translated and instrumented with low-level STM API constructs for the purpose of conflict detection and potential rollback. Instrumenting calls to a precompiled function proceeds as follows:

i. Log Values of Thread-local Variables. For every thread-local variable in the write clause, the compiler uses the log functions to log their values as they may be needed in case of a rollback.

ii. Create Local Copies for Shared Variables. For every shared scalar variable in the read and write clauses, the compiler creates a local variable on the stack. The compiler initializes the local copies of the variables in the read clause by using `stmRead⟨Type⟩`. For other shared variables (such as arrays, objects, and dynamic data structures) in the read and write clauses, the compiler uses `malloc` to allocate space for their local copies and assigns the starting addresses to the corresponding pointers. Function `stmReadBytes` is called to copy data for these variables. The compiler replaces the function arguments with their local copies.

iii. Update Values of Shared Variables. After the function is completed, the shared variables in the write clauses need to be updated with the values in their local copies. Therefore, the compiler inserts calls to `stmWrite⟨Type⟩` and `stmWriteBytes` after the function call for shared variables in the write clause. Finally, the compiler frees all temporarily-allocated variables.

The code below shows the translated call to precompiled function `memcpy`, whose annotation was described before.

```
int l_num = stmReadInt(tx, &num);
void *l_src = (void*)malloc(tx, 256);
void *l_dst = (void*)malloc(tx, 256);
stmReadBytes(tx, (void*)l_src, (void*)src, 256);
memcpy(l_dst, l_src, l_num);
stmWriteBytes(tx, (void*)dst, (void*)l_dst, 256);
free(l_src); free(l_dst);
```

B. Irreversible Functions

In some transactions that call irreversible functions, other statements may not be data-dependent on the irreversible functions. For example, function `printf` only prints text on the screen but does not produce any data. Another example is function `system`, which invokes the shell to execute a system command. Other statements in the transaction may not depend on the system command. Therefore, in

these transactions, the execution of such functions can be safely suspended during speculative execution. We store the input of these functions during the speculative execution and execute them when the transactions have succeeded.

The construct `#pragma tm suspend` is designed to annotate the functions to be suspended during the speculative execution. Its syntax is similar to that of the `precompiled` construct except that there is no `write` clause; it annotates function declarations, as follows:

```
#pragma tm suspend read(c)
int putchar (int c);
```

The `putchar` function is an output function from the standard C library—it prints the character `c` to the current position in the standard output. Since the function does not use any pointer as argument, we can annotate it when it is declared. Similar to the `precompiled` construct, the `suspend` construct can also be used at function call sites as shown below.

```
char str[256];
#pragma tm transaction
{
  ... // statements here
  #pragma tm suspend read(str, (*str, 256))
  puts(str);
}
```

In the above example, function `puts` writes the string pointed to by `str` to standard output. As the string ends in a null character, we do not know the string length at the function declaration. We solve this by annotating the function at its call site, where the maximum length of the string is known.

Calls to an annotated irreversible function need to be translated and instrumented to enable suspending with transactions. Our compiler instruments calls to irreversible functions as follows:

i. Record Arguments. For every variable in the read clause, the compiler pushes its value in the thread-local queue `args`. For shared variables, `stmRead⟨Type⟩` or `stmReadBytes` is called inside the push function to ensure consistency. These values will be used to invoke the irreversible function outside the transaction.

ii. Record Function. The compiler replaces the function call with a statement that saves the function identifier (generated from the function name for each annotated function) in the thread-local queue `funcs`. The function identifier will be used to invoke the function outside the transaction.

iii. Call Function Outside. The compiler generates a wrapper function `resume_suspended_funcs` that goes through these queues and calls the irreversible functions they contain. The wrapper function will be called from `stmEnd` when the transaction is successfully committed.

The code below shows the translated call to irreversible function `putchar`, whose annotation was described before.

```
stmBegin(tx);
... // statements here
args.sharedpush(tx, &c, sizeof(char));
funcs.push( F_PUTCHAR );
stmEnd(tx);
...
void resume_suspended_funcs() {
  while ( !funcs.empty() )
    switch ( funcs.pop() ) {
      case F_PUTCHAR:
        putchar( *((char*)args.pop()) );
        break;
      ...
    }
}
```

We use a global lock to prevent the wrapper functions called in different transactions from interleaving. A thread needs to acquire the lock before performing a wrapper function during the commit phase.

Our `precompiled` and `suspend` constructs can be used to annotate most C/C++ standard library functions. However, there is one case where these constructs cannot be used—the standard template library (STL). This is because some STL functions operate on linked data structures, hence it is difficult to determine the memory locations that are accessed prior to executing them.

IV. Optimizations

In Section II-B2 we have presented our static data race analysis that helps reduce the number of read/write barriers inserted in the transactional code. In this section, we present three other compile time optimizations to reduce the time overhead incurred by STM.

A. Eliminating Redundant Barriers

During code instrumentation, redundant read/write barriers may be introduced in the code, which can significantly increase the STM overhead. The code below shows two arithmetic statements and their intermediate code with read/write barriers.

Original code	*Intermediate code*
`b = a+1;`	`stmWriteInt(tx, &b, \` `  stmReadInt(tx, &a)+1);`

`b = a*b;`	`stmWriteInt(tx, &b, \`
	`  stmReadInt(tx, &a)* \`
	`  stmReadInt(tx, &b));`

In the intermediate code, `read barrier` is called three times and `write barrier` is called twice. Actually, only two barriers are required in this code, one read barrier for `a` and one write barrier for `b` (all other barriers are redundant).

To eliminate redundant barriers in an expression, our compiler first creates temporary variables to hold the values loaded/stored by read/write barriers and uses the temporary variables in the expression. The code below shows the previous intermediate code with temporary variables inserted for read/write barriers.

```
l_a = stmReadInt(tx, &a);
l_b = l_a+1;
stmWriteInt(tx, &b, l_b);
l_a = stmReadInt(tx, &a);
l_b = stmReadInt(tx, &b);
l_b = l_a*l_b;
stmWriteInt(tx, &b, l_b);
```

With temporary variables, read/write barriers are separated from the original statements. A read barrier can be eliminated if it is pre-dominated by read or write barriers to the same variable within the same transaction. A write barrier can be eliminated if it is post-dominated by write barriers to the same variable within the same transaction. The code below shows the final code generated for the previous example after elimination of redundant barriers.

```
l_a = stmReadInt(tx, &a);
l_b = l_a+1;
l_b = l_a*l_b;
stmWriteInt(tx, &b, l_b);
```

B. Reducing Searches in Write Buffers

Since the overhead of write buffering-based STMs comes largely from searching write buffers [12], eliminating unnecessary data searches in write buffers can greatly improve performance. To eliminate unnecessary searches in write buffers, our compiler checks the control flow for each read barrier. If a read barrier is not preceded by any write barrier to the same variable within the same transaction, it does not need to search the transaction's write-set since the variable cannot be in the write-set. Our compiler replaces such read barriers with calls to `stmDirectRead⟨Type⟩`, which is a read barrier that reads a value in a regular memory location without searching the write-set.

C. Synchronization Between Transactions

Speculative parallelization often requires transactions to be committed in a specific order, to preserve the sequential semantics of the original program. One way to enforce a commit order between transactions is to place synchronization code prior to the end of the transaction. However, this has two major drawbacks. First, performing synchronization in transactions introduces extra overhead since synchronization code usually does not need to be executed speculatively. Second, performing synchronization in transactions may cause transactions to abort, due to inconsistent states of shared data structures used for synchronization. Therefore, it increases transaction abort rates. For example, in the following code, busy-waiting is used to synchronize the transaction commit.

```
#pragma tm transaction
{
  ... // statements here
  while (signal == 0);
}
```

In the above example, the transaction will not commit until a signal is received. Let us assume that when the transaction enters the busy-waiting loop, variable `signal`'s value is 0. The value of `signal` can be changed to 1 after the transaction has started, e.g., by another transaction. This leads to an inconsistent memory state since the transaction eventually sees two values ('0' and '1') for variable `signal`. Most STM implementations, such as Transactional Lock II (TL2) [3], will abort the transaction in this case since inconsistent memory state may trigger a fatal exception (e.g., segmentation fault) or cause the transaction to enter an endless loop.

We introduce an STM construct, `#pragma tm beforevalidate`, which specifies code that is executed non-speculatively *immediately before* a transaction (specified via `#pragma tm transaction`) is validated and committed. The `beforevalidate` construct is designed for synchronization between transaction commits to keep the sequential semantics of the original program. For example, the preceding synchronization code can be written as follows:

```
#pragma tm transaction
{
  ... // statements here
}
#pragma tm beforevalidate
{
  while (signal == 0);
}
```

In the example, the busy-waiting loop is executed non-speculatively before transaction validation and commit. Therefore, it will not cause extra overhead or increase transaction abort rate.

V. Evaluation

This section evaluates our approach. Our compiler is implemented on top of ROSE [13], an open source compiler infrastructure. We use the built-in call graph generator in ROSE [13] to generate the static call graph. The static data race detector, Locksmith [16], is used to find potentially shared variables. Since Locksmith is designed for C, we use the Comeau C++ compiler [18] to convert C++ code to C-like code, which is then passed to Locksmith. We used our compiler implementation with the TL2 STM library [3], though it can target any write-buffering based STM system that supports the low-level TM interface listed in Table III.

The experiments were conducted on a DELL PowerEdge T605 machine (Intel Xeon 8-core 2.0GHz) running *CentOS v5.5*. We performed experiments on 10 applications: eight STAMP benchmarks and two real applications—Velvet and ITI.

A. STAMP Benchmarks

STAMP is a benchmark suite designed for TM research. It consists of eight parallel benchmarks [5]. All STAMP benchmarks are originally instrumented with low-level TM API constructs. To evaluate our approach, we replaced the low-level TM API constructs in the STAMP benchmarks with our programming constructs. Each benchmark has two input data sets. In our experiments, we used the larger input data set, which is more suitable for experiments on real machines.

1) Programming Effort: To compare the programming effort with prior approaches, Table IV shows the number of programming constructs inserted into the benchmarks, using the low-level STM API, the Intel STM compiler, and our approach. The reported numbers include the number

Program	Programming effort					
	Low-level API		Intel	*Our compiler*		
	barrier	other		bound.	lib.	irrever.
Bayes	49	127	75	15	0	0
Labyrinth	36	62	31	3	0	0
Genome	60	62	31	5	0	0
Intruder	75	190	95	3	0	0
Kmeans	8	9	3	3	0	0
Ssca2	24	31	10	10	0	0
Vacation	63	296	151	3	0	0
Yada	99	225	115	6	0	0
Velvet	–	–	–	2	0	9
ITI	–	–	–	1	2	30

Table IV
PROGRAMMING EFFORT (NUMBER OF CONSTRUCTS) WHEN USING THE LOW-LEVEL STM API, THE INTEL STM COMPILER, AND OUR COMPILER. PROGRAMMING EFFORT USING THE LOW-LEVEL STM API IS SPLIT INTO CONSTRUCTS USED FOR READ/WRITE BARRIERS AND OTHERS, WHILE PROGRAMMING EFFORT USING OUR COMPILER IS SPLIT INTO CONSTRUCTS USED FOR TRANSACTION BOUNDARIES, PRECOMPILED FUNCTIONS, AND IRREVERSIBLE FUNCTIONS.

of programming constructs inserted in the library functions called in these benchmarks.

Compared to the low-level STM API, our approach requires on average 97% fewer programming constructs to be inserted into each benchmark. For each benchmark in the PARSEC benchmark suite, we counted the number of accesses to global/heap variables within transactions. We found that on average each transaction contains 144 accesses to global/heap variables, out of which 81 are shared accesses. To achieve better performance with the low-level STM API, programmers need not only place read/write barriers, but also put effort in examining all the accesses to global/heap variables to see whether they are shared accesses. We liberate programmers from the burden of identifying shared accesses and placing read/write barriers. Compared to the Intel STM compiler, our approach requires on average 91% fewer programming constructs to be inserted into each benchmark. With our approach, programmers do not need to annotate any function called within transactions. *Overall, our approach only requires 3–15 programming constructs for each benchmark.*

2) Performance: Figure 2 shows the speedups achieved for each STAMP benchmark using different numbers of threads. For each benchmark, the baseline is the single-threaded original version of the program. Numbers greater than 1 reflect better performance than the single-threaded original versions. The numbers are measured using all of our optimizations. With 8 threads, our approach achieves speedups for all STAMP benchmarks except `vacation-high` and `yada`. For `vacation-high` and `yada`, the large number of read/write barriers imposes a significant performance penalty. *On average, our approach achieves* 1.65x *speedup for the STAMP benchmarks with 8 threads.*

3) Optimization Benefits: This section evaluates the benefits of various optimizations proposed in this paper. The optimizations include using static data race analysis (DRA) to avoid placing barriers for accesses that do not cause data races, caching the values loaded/stored by shared reads/writes to eliminate redundant barriers (ERB), and eliminating unnecessary searches (EUS) in write buffers for data that are definitely not present in write buffers.

Program	Baseline	DRA	DRA+ERB
Bayes	172	87	71
Labyrinth	107	67	67
Genome	40	40	37
Intruder	183	152	148
Kmeans	7	7	7
Ssca2	24	24	24
Vacation	196	180	156
Yada	424	304	289
Average	*144*	*108*	*100*

Table V
NUMBER OF READ/WRITE BARRIERS INSERTED WITH VARIOUS OPTIMIZATIONS.

Table V compares the number of read/write barriers inserted with different optimizations. In the baseline, we inserted read/write barriers for every access to global/heap data, which are potentially shared by threads. As we can see, in the baseline, each benchmark has on average 144 read/write barriers. DRA reduces the number of barriers by 25.3% on average for these benchmarks. The number of barriers inserted in `yada` is reduced significantly since `yada` operates on numerous locally-allocated objects. ERB further reduces the number of barriers by 7.2%. EUS does not reduce the number of barriers since it only eliminates the unnecessary searches in write buffers. Since `kmeans` and `ssca2` have only small transactions that do not call any functions, their baseline does not contain any barriers to eliminate. *Overall, our optimizations eliminate* 30.7% *of the barriers from the baseline.*

Figure 3 shows the impact of various optimizations on program performance. The numbers were measured using 8 threads. In the baseline, read/write barriers are inserted for every access to global/heap data. DRA improves the performance by 23.3% on average for these benchmarks. The performance of `bayes` and `labyrinth` is improved most since a lot of barriers in them are eliminated by DRA. ERB and EUS further improve the performance by 7.4% and 5.4% for these benchmarks. EUS significantly improves performance for `genome` and `kmeans` since they have shared read-only data. *Overall, our optimizations improve the performance by* 32.8% *on average for these benchmarks.*

4) Comparison: Figure 4 compares the performance of hand-coded transactional code using the low-level STM API, the Intel STM compiler, and our approach. The STAMP benchmark suite provides hand-coded transactional code via low-level STM API constructs. To measure the performance of the Intel STM compiler, we apply their programming constructs to the STAMP benchmarks. The numbers were

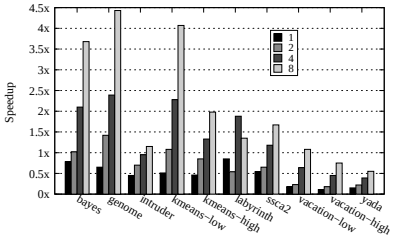

Figure 2. Speedups over sequential versions of the same programs achieved by our compiler using 1, 2, 4, and 8 threads.

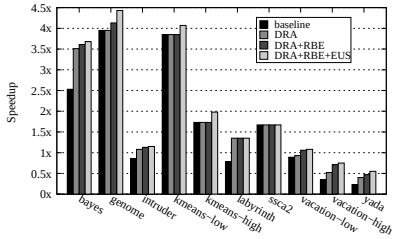

Figure 3. Performance with different optimizations.

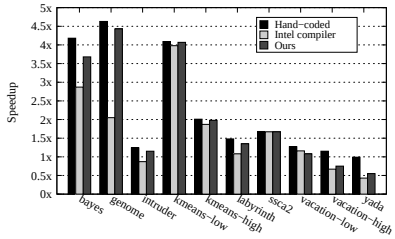

Figure 4. Performance comparison of the hand-coded transactional code, the Intel STM compiler, and our compiler.

measured using 8 threads. Among the three implementations, the hand-coded transactional code achieves the best performance. This is expected, since a lot of effort has been spent on manually optimizing the hand-coded transactional code provided by STAMP. For example, there are no redundant barriers in the hand-coded transactional code. The performance of our approach is quite close to that of the hand-coded transactional code. *Compared to the hand-coded transactional code, the programs generated by our compiler are slower by* 14.3%. *Compared to the Intel STM compiler, our approach improves the performance by* 20.8%.

B. Real Applications

This section evaluates our approach using two real applications, `Velvet` and `ITI`. We use these applications to show that our approach has low programmer burden and can improve performance for loops with precompiled and irreversible functions. The low programming burden is evident in the last two rows of Table IV: we had to add just 11 constructs for Velvet and 33 for ITI. Note that, low-level STM APIs and the Intel STM compiler cannot be used due to the presence of precompiled and irreversible functions.

1) Velvet: `Velvet` [10] is a widely-used genomic assembler designed for short read sequencing technologies. Due to its popularity, developers have put a lot of effort into parallelizing it. In the latest version of `Velvet` (version 1.1), fine-grained locks (i.e., one lock for each shared object) are used to parallelize two loops in function `fillUpGraph`, which account for approximately 50% of the execution time. To apply fine-grained locks, 257 lock-related statements were added by the `Velvet` developers. Apart from the extra code, the programmer must ensure that there is no deadlock and no livelock. Lock contention needs to be managed to maximize program performance. For example, programmers must determine whether threads should spin or block when acquiring a lock.

We applied our approach to the two loops. Each iteration of the two loops is treated as a transaction. Numerous output operations (e.g., `fprintf`) and system calls (e.g., `sysconf`) are called inside the two loops. Since these functions are irreversible, we inserted 9 programming constructs to suspend and perform their execution outside the transactions. In total, we inserted 11 programming constructs in `Velvet`.

Figure 5 compares the performance of our approach and fine-grained locks (FGL). The baseline is the sequential version of `Velvet`. The numbers were measured using nucleotide sequence *SRR027005* [19] as input. With only 11 programming constructs added, our approach achieves 1.48x speedup using 8 threads. Compared to code with fine-grained locks, the code generated by our compiler is slower by only 11.4%. Considering the significantly lower programming effort, our approach provides an easy way to parallelize real applications.

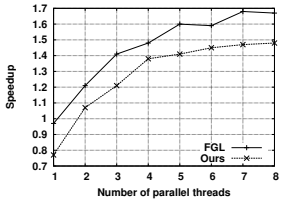

Figure 5. Speedups of Velvet over its sequential version.

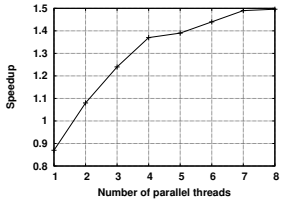

Figure 6. Speedup of ITI over its sequential version.

2) Incremental Tree Inducer: Incremental tree inducer (ITI) [11], also called Direct Metric Tree Induction (DMTI), is a widely-used decision tree constructor; it incrementally constructs decision trees from labeled examples. The application has not been parallelized before.

We applied our approach to the main loop, each iteration of which reads a labeled example and updates the decision tree. The loop body is annotated as a transaction. Both precompiled and irreversible library functions are used inside the loop. Two C string functions, `strlen` and `strcmp`, are called inside the loop. Since the string length is known, we used 2 `precompiled` constructs to annotate their declarations. C standard output function, `printf`, is also called inside the loop. Since different calls to `printf` use different formats of arguments, we inserted 30 `suspend` constructs to annotate its call sites. In total, we added 33 programming constructs.

Figure 6 shows the speedups achieved by our approach for `ITI`. The baseline is the single-threaded original version of `ITI`. We used data set `agaricus lepiota` [20] as input. We achieved a 1.50x speedup using 8 threads, which demonstrates that our approach can be used to parallelize real applications that contain both precompiled and irreversible functions.

VI. Related Work

Besides the Intel STM Prototype Compiler [6], some other works have also introduced basic TM constructs into C or C++. Tanger [21] is another STM compiler for C/C++. Similar to the Intel STM compiler, it does not provide STM constructs for synchronization, precompiled library functions or irreversible functions. It relies on dynamic instrumentation to support precompiled library functions in transactions. Luchangco et al. [22] theoretically analyzed different design options for integrating STM into C++ without doing any real implementation. OpenTM [9] is an extension to OpenMP [23] and focus more on the expression of loop-level speculative parallelism based on TMs. Milovanović et al. [24] proposed another extension to OpenMP. It supports a multithreaded STM design with a dedicated thread for eager asynchronous conflict detection. Performing conflict detection in a separate thread saves time in other threads. SpiceC [25] is a programming model for loop-level parallelization. It is specially designed to support multiple forms of parallelisms, including speculative parallelism.

Many works have introduced TM constructs into managed languages. AtomCaml [26] introduced first-class constructs to support atomic execution of code written in Objective Caml, which is based on a uniprocessor execution model. Adl-Tabatabai et al. [14] presented compiler and runtime optimizations for TM constructs in JAVA. Their system supports composition of transactions and partial roll back. They use just-in-time (JIT) optimizations on STM operations. Hindman and Grossman [15] developed a source-to-source translator to support atomicity in JAVA. Their implementation is based on locks.

VII. Conclusion

We have presented an approach for programming speculatively-executed code in C/C++. Our proposed programming constructs only require programmers to annotate transaction boundaries. Our constructs also provide support to enable parallel execution of transactions that contain precompiled and irreversible library functions. We have evaluated our compiler implementation. Compared to the low-level STM API, our approach requires 97% fewer programming constructs to be inserted into the STAMP benchmarks. When using 8 threads, our compiler implementation achieves a 1.62x average speedup for the ten applications.

References

[1] L. Rauchwerger and D. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," in *PLDI*, 1995, pp. 218–232.

[2] N. Shavit and D. Touitou, "Software transactional memory," in *PODC*, 1995, pp. 204–213.

[3] D. Dice, O. Shalev, and N. Shavit, "Transactional lock II," in *Distributed Computing*, 2006, pp. 194–208.

[4] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," in *PPoPP*, 2008, pp. 237–246.

[5] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IISWC*, 2008, pp. 35–46.

[6] "Intel C++ STM compiler, prototype edition," http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/.

[7] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai, "Code generation and optimization for transactional memory constructs in an unmanaged language," in *CGO*, 2007, pp. 34–48.

[8] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian, "Design and implementation of transactional constructs for c/c++," in *OOPSLA*, 2008, pp. 195–212.

[9] W. Baek, C. C. Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun, "The opentm transactional application programming interface," in *PACT*, 2007, pp. 376–387.

[10] D. R. Zerbino and E. Birney, "Velvet: algorithms for de novo short read assembly using de bruijn graphs," *Genome Research*, vol. 18, pp. 821–829, 2008.

[11] P. E. Utgoff, N. C. Berkman, and J. A. Clouse, "Decision tree induction based on efficient tree restructuring," *Mach. Learn.*, vol. 29, no. 1, pp. 5–44, October 1997.

[12] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg, "McRT-STM: a high performance software transactional memory system for a multi-core runtime," in *PPoPP*, 2006, pp. 187–197.

[13] D. Quinlan, "Rose: Compiler support for object-oriented framework," in *CPC*, 2000.

[14] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman, "Compiler and runtime support for efficient software transactional memory," in *PLDI*, 2006, pp. 26–37.

[15] B. Hindman and D. Grossman, "Atomicity via source-to-source translation," in *MSPC*, 2006, pp. 82–91.

[16] P. Pratikakis, J. S. Foster, and M. Hicks, "LOCKSMITH: Practical static race detection for C," *TOPLAS*, vol. 33, no. 1, pp. 3:1–3:55, January 2011.

[17] L. Van Put, D. Chanet, B. De Bus, B. De Sutler, and K. De Bosschere, "DIABLO: a reliable, retargetable and extensible link-time rewriting framework," in *ISSPIT*, 2006, pp. 7–12.

[18] "Comeau C++ compiler," http://www.comeaucomputing.com/.

[19] "DDBJ sequence read archive," http://trace.ddbj.nig.ac.jp/dra/index_e.shtml.

[20] "UCI machine learning repository," http://archive.ics.uci.edu/ml/.

[21] P. Felber, T. Riegel, C. Fetzer, M. S. kraut, U. Müller, and H. Sturzrehm, "Transactifying applications using an open compiler framework," in *TRANSACT*, 2007.

[22] V. Luchangco, L. Crowl, Y. Lev, D. Nussbaum, and M. Moir, "Integrating transactional memory into C++," in *TRANSACT*, 2007.

[23] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *IEEE computational science & engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[24] M. Milovanović, R. Ferrer, V. Gajinov, O. S. Unsal, A. Cristal, E. Ayguadé, and M. Valero, "Multithreaded software transactional memory and OpenMP," in *MEDEA*, 2007, pp. 81–88.

[25] M. Feng, R. Gupta, and Y. Hu, "SpiceC: scalable parallelism via implicit copying and explicit commit," in *PPoPP*, 2011, pp. 69–80.

[26] M. F. Ringenburg and D. Grossman, "Atomcaml: first-class atomicity via rollback," in *ICFP*, 2005, pp. 92–104.