Supporting Legacy Binary Code in a Software Transaction Compiler with Dynamic Binary Translation and Optimization

Cheng Wang, Victor Ying, and Youfeng Wu

Programming System Lab
Microprocess Technology Labs
Intel Corporation
2200 Mission College Blvd.
Santa Clara, CA 95052, USA
{cheng.c.wang,victor.ying,youfeng.wu}@intel.com

Abstract. Transactional memory (TM) has been shown to be a promising programming model for multi-core systems. We developed a Software-based Transactional Memory (STM) compiler that generates efficient transactional code for transactions to run on a STM runtime without the need of transactional hardware support. Since real-world applications often invoke third party libraries available only in binary form, it is imperative for our STM compiler to support legacy binary functions and provide an efficient solution to convert those invoked inside transactions to the corresponding transactional code. Our STM compiler employs a Lightweight Dynamic Binary Translation and Optimization Module (LDBTOM) to automatically convert legacy binary functions to transactional code. In this paper, we describe our LDBTOM system, which 1) seamlessly integrates the translated code with the STM compiler generated code to run on the STM runtime, and 2) optimizes the translated code taking advantage of dynamic optimization opportunities and STM runtime information. Although the binary code is inherently harder to optimize than high-level source code, our experiment shows that it can be translated and optimized into efficient transactional code by LDBTOM.

1 Introduction

Transactional memory [3][11][12][15][16] provides a powerful programming model to design concurrent programs. This model guarantees a large region of code, i.e. a transaction, to be executed atomically, and thus enables ordinary programmers to write correct and efficient concurrent applications without using locks. Many difficult issues associated with lock-based programs, such as deadlock, non-scalable composition, priority inversion, can thus be alleviated or eliminated.

To support Software Transactional Memory (STM [1][7][11][19]) or hardware assisted software transactional memory (HASTM [2]), memory operations inside a transaction need to be augmented with STM runtime API calls to check for

L. Hendren (Ed.): CC 2008, LNCS 4959, pp. 291–306, 2008.

concurrency conflicts and log for transaction rollback. Existing STM compiler [7] for C/C++ programs can automatically insert the API calls to STM runtime (called barriers) for memory operations inside the transactions, if the program source code is available. Unfortunately, real-world applications often invoke third party libraries available only in binary form not compiled by a STM compiler. It is imperative for STM compilers to allow legacy binary functions to be called inside transactions. Otherwise, the usability of STM would be greatly limited. For example, a transaction may need to call a "*qsort*" library function. If library functions are not allowed to be called inside a transaction, the user would have to re-implement it in source code and compile it with the STM compiler.

We will refer to non-transactional library code the "legacy code". There are a number of approaches to support legacy code in STM compiler. One way is for the library supplier to provide a separate transactional version of the library. This not only places a significant burden on the library development and validations, but also poses engineering issues with managing multiple versions of the same library. There are also old libraries that cannot be recompiled. Another way is to statically translate library routines called in a transaction to transactional version. However, dynamically linked library routines may not be available at compiler time. Even if the library code is available at compile-time, static translation is known to be limited and may not be able to translate certain routines when indirect branches or calls are present. It is also possible to use special hardware to trap memory load and store operations so that the barrier operations can be executed for these operations. That approach, however, increases hardware costs on die area and power consumption. Our STM compiler [7] employs a Lightweight Dynamic Binary Translation and Optimization Module (LDBTOM) [8] to automatically convert the legacy binary functions to transactional code if they are called inside transactions at runtime.

There are many existing works on STM runtime systems [12][13][19], STM compilers [1][7][11] and Dynamic Binary Translation techniques [4][8][8][9][17]. However, to seamlessly integrate these techniques together for legacy binary code support in software transactional memory brings challenges. For example, a normal dynamic binary translator translates all the binary code in the whole application. But in our case, LDBTOM must only translate the legacy code, and must not translate the transactional code generated by STM compiler. Translation of the transactional code generated by the STM compiler not only slows down the program, but may also cause severe problems, such as livelocks [5], in the STM runtime. Therefore, we need close collaborations between the dynamic binary translation and STM compiler to distinguish the legacy binary code from the transactional code generated by STM compiler. Sophisticated program analysis may help identify legacy function calls inside transactions, but it is capable of completely solving the problem, especially in the present of indirect function calls and indirect branches. As another example, while the compiler generated code may be shared among different threads, most dynamic binary translation systems (including our DBT system) implement thread-private code cache for its simplicity of implementation. The efficient linking between thread-shared compiler-generated code and the thread-private DBT-generated code is a challenging issue. Furthermore, most of stack memory are private to the thread and thus do not need memory conflict checking. However, there are situations where a stack location may be shared among multiple threads and to generate efficient and correct code for stack references inside transactions poses challenges for dynamic binary translation.

On the other hand, LDBTOM can optimize the translated code taking advantage of dynamic optimization opportunities. For example, the dynamic optimization allows us to generate efficient STM code based on the assumption that stack variables are not shared among different threads. In case the program execution makes it possible for stack variables in one thread accessible to other threads, we can flush and regenerate new STM code conservatively before the stack variable sharing actually happens.

Overall, this paper makes the following major contributions.

- We develop a seamless framework to support legacy library code in a STM compilation system.
- We provide a complete solution to make sure that no legacy code will be executed without translation and no compiler generated transactional code are translated, even when the static type checking fails to detect user's programming error. We also provide efficient solutions for linking between the thread-shared compiler-generated code and the thread-private DBT-generated code.
- We developed a number of dynamic optimization techniques, such as stack variable filtering, dead register and conditional code (*%eflags*) saving elimination, redundant barrier elimination, inlining, etc, to dramatically reduce the overhead of translated code for transactional execution.
- We provide experimental result comparing translated code vs. STM compiler generated code using SPLASH-2 benchmark and a set of concurrent data structure benchmarks. Even though legacy binary code usually has very little high-level information available for sophisticated analysis and optimizations, our experimental results on SPLASH-2 benchmark shows that LDBTOM only causes about 1% overhead over compiler optimized STM code. Even for the data structures benchmarks which spend almost all their execution time inside legacy functions called inside transactions (to stress test the LDBTOM overhead), the LDBTOM optimized code runs only about 80% slower than the hand-optimized STM code. In contrast, straightforward translation would perform more than 8 times slower than the hand-optimized code.

The rest of the paper is organized as follows. Section 0 discusses the related work. Section 0 overviews the STM compiler. Section 0 describes LDBTOM infrastructure and transformations. Section 0 discusses optimization strategies. Section 0 provides experimental results. Section 0 concludes the paper and points out future research directions.

2 Related Work and Issues

Different kinds of hardware transactional memory mechanisms are described in [3][16][18]. Software transactional memory was introduced in [12][19]. Efficient implementations of STM in managed environment were provided in [1][11]. An efficient implementation of STM in unmanaged environment was developed in [7]. Hybrid transactional memory [14] proposes to combine HW and SW transactional memory implementations.

Dynamic binary translation has been an active research topic in recent years. It has be used to support backward compatibility [4], enhance reliability [8] and security [9], reduce power consumption [17], as well as improve programmability, as reported in this paper, to support transactional memory.

JudoSTM [14] annotates the program source code to specify transactions, and implements a transactional memory system through dynamic binary rewriting. With program source code available, dynamic binary rewriting for the whole transactions seems less efficient. Our LDBTOM takes advantages of sophisticated static compilation and only dynamicaly translates and optimizes the legacy code in transactions.

There are a number of open issues remaining to support legacy binary code in STM. Transactions need to support rollback in case a conflict is detected. The users must ensure that the codes insides transactions do not perform operations that cannot be rolled back, such as the system calls and input/output operations. System call and I/O issues are being actively addressed in research community with open nested transaction [13] and restricted transaction [6]. These are general issues for transactional memory and are not unique to legacy code so we consider them beyond the scope of this paper.

There is also the issue with software implemented synchronization operations (e.g. lock, barrier) in legacy code. Lock based programs can use different locks for different critical sections, so that critical sections controlled by different locks do not necessarily synchronize with each other, even when there are conflict memory accesses among them. But transactions are required to synchronize with each other whenever there are conflict memory accesses among them, just as if all of them are controlled by a single lock. Thus the single-lock semantics for transactions is different from the semantics of the individually locked sections, and straightforwardly converting locked sections to transactions may result in deadlock [5]. Lock-based programs may also implement barriers to join multiple threads together, while transaction can not accomplish the barrier synchronizations easily. These are general issues with converting lock-based program to transactions, no matter the locks are used in source code or legacy binary code. We conducted a preliminary study to address this issue in a separate paper [20].

In this paper, we assume that the user will decide that a library routine does not invoke system calls, I/O operations, and locked-sections, and can be called inside a transaction. There are significant portions of legacy code, e.g. majority of routines provided in LIBC and LIBM, etc, that can be safely called inside transactions as long as they are translated to check for conflict and log operations for rollback. This paper targets this portion of libraries. LDBTOM currently report runtime errors when I/O and system calls in legacy code are translated.

3 STM Compiler

Our STM compiler [7] targets C/C++ programs. The compiler provides programming language constructs to write programs with transactions. An example code illustrating the transaction constructs is shown in Fig. 1. The *tm_atomic* pragma specifies that the statement (usually a block statement) following it is a transaction (an

atomic operation). If a conflict is detected during the execution of a transaction, the state of the transaction will be rolled back to the same as before the transaction execution and the transaction is re-executed. Transactions can also be nested. We support closed nested transactions [13]. In our implementation, when a conflict is detected in a nested transaction, the outmost transaction is aborted. If a *tm_abort()* intrinsic operation is executed inside a nested transaction, the innermost transaction is rolled back. When an inner transaction completes, its memory updates are not actually committed until the out-most transaction commits. The *tm_commit()* intrinsic operation explicitly commits a transaction before it reaches the end.

The *tm_function* pragma specifies a transactional function (e.g. *foo()*) that may be called inside a transaction. For each transactional function, the STM compiler creates a transactional clone and calls it inside transactions while calls the original function outside transactions. For a function called inside a transaction that is not specified as *tm_function*, the compiler will treat it as a legacy function and generate code to convert the function to transactional at runtime. For the example in Fig. 1, the function *qsort()* is not declared as a *tm_function* and is converted to a transactional routine at runtime.

```
#pragma tm_function
int foo (int) {…}

extern int qsort(…);
…
#pragma tm_atomic
{
    stmt1
    foo(3); // call TM clone of foo
    if (cond1) {
        tm_commit();
        return;
    }
    #pragma tm_atomic
    {
        stmt2;
        tm_abort();
    }
    qsort(); // translated at runtime
    stmt3
}
```

Fig. 1. Transaction construct for STM compiler

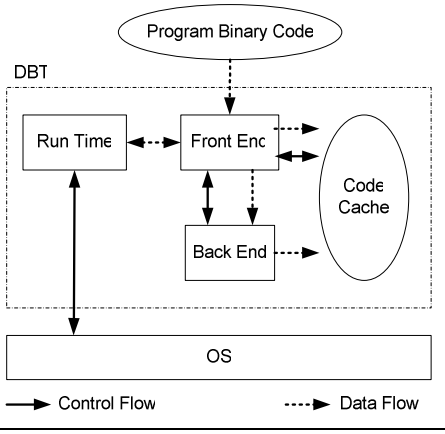


Fig. 2. Dynamic binary translator structure

In order to execute a transaction atomically, the STM compiler needs to insert calls to STM runtime routines (called barriers) for memory read and write operations inside the transaction to detect conflict with the memory accesses in other transactions. The read and write barriers first make sure no other transaction currently owns the data being accessed. The write barrier then acquires ownership for the data being updated,

updates the data in place, and keeps the old value in an undo-log in case the transaction has to rollback. The read barrier logs the data version so it can be checked at transaction commit time to make sure that the set of data read in the current transaction is consistent with respect to the committed transactions. In addition to the barriers, the STM compiler also generates code to checkpoint the live-in values at the beginning of the transaction, rollback transaction to the initial state when a conflict is detected or when a *tm_abort()* intrinsic is invoked, and commit the transaction results at the end of the transaction.

The primary overhead for STM compiler generated code is the barrier operations. To reduce the overhead, the STM compiler generates inlined barrier code to reduce function call overheads. It also uses compiler analysis to determine thread-private references that can never conflict with accesses in other transactions so as to omit their barriers. Furthermore, the STM compiler eliminates redundant barriers for data accesses with the same addresses. With these optimizations, the STM compiler generated code performs comparative to hand-optimized code and is competitive to fine-grain lock-based code, and much better than coarse-grain locking version when running on multiple processors [7].

4 Compiler Integration with DBT

For a library function called inside a transaction, the STM compiler invokes LDBTOM to convert it to a transactional function at runtime. LDBTOM is based on our Dynamic Binary Translation (DBT) research framework: StarDBT [8]. The overall structure of StarDBT is shown in Fig. 2. The DBT runs on top of OS as a user-level run-time system. The program binary code is dynamically translated and stored into the code cache. The translated code can be executed under the control of the DBT, which allows us to apply different dynamic binary translation techniques to the code.

The DBT consists of three individual modules: the *Runtime* module, the *Frontend* module and the *Backend* module. The *Runtime* module provides the system supports for the DBT. The *Frontend* module manages the execution for dynamic binary translation. It dynamically recognizes the original program instructions, translates them into instructions in the code cache, and controls the code execution in the code cache. The *Frontend* module also collects program profiling information during the code execution and selects hot traces based on the profiling information for run-time optimization by the *Backend*. The *Backend* module performs run-time optimization for the dynamic binary translations. It builds an intermediate representation (IR) from the hot traces selected by the *Frontend* module. It then performs optimizations on the IR, and finally generates optimized code into the code cache to improve performance.

StarDBT has been used for many research works to improve reliability [9], enhance security [10], etc. In this work, we leverage a simplified module from StarDBT called LDBTOM, to support legacy binary code in STM. The compiler invokes LDBTOM directly in the generated code with the address of the library function passed as a argument for translation. LDBTOM starts to translate the code at this address and runs the translated code from the translation code cache.

When a translated library function returns to a transaction or calls back to an STM compiler generated function, the execution should continue to the compiler generated transactional code without translation. For this purpose, the STM compiler adds all the entry points in the compiler generated code to LDBTOM's runtime dispatch table, including returning points for library functions called in transactions. If LDBTOM finds the branch/call target in the dispatch table, the branch/call will directly connect to the targeted code. Otherwise the code at the target will be translated before execution. Therefore, the added entry points in the dispatch table will naturally cause LDBTOM to go back to the compiler generated STM code for execution.

Once a library function is translated, subsequent calls to the same function should not invoke LDBTOM again. To accomplish this, STM compiler builds a table, called Translation Linkage Table (TLT), similar to the Procedure Linkage Table for shared libraries, with one entry per static call site that may call a legacy library function in STM compiler generated code. Each call to a library function is generated as an indirect call through the corresponding entry in the TLT. The first time a library function is called, its TLT entry contains LDBTOM's entry point to translate the legacy function. After that, the TLT entry contains the starting address of the translated function, and late call to the function is mostly an indirect call through the TLT without translation. This scheme works well for direct calls to library functions. For an indirect call, however, the library function pointed to by the function pointer may change at runtime. Consequently, we cannot simply place the starting address of the translated function in the TLT entry. Instead, we place a dispatch table lookup routine in that entry. The lookup routine searches the dispatch table to determine if the current function pointer is to an already-translated function. If so, the translated function is directly called. Otherwise, LDBTOM is invoked to translate the indirectly called function. The TLT entry for an indirect call may also be expanded to include a few conditional branches for those frequently called targets.

LDBTOM uses a thread-private code cache for each user thread. For a parallel loop body, the same static code is executed by multiple threads, and the code needs to be compiled to work with code in multiple code caches in different threads. Our STM compiler creates a thread-private TLT for each thread and uses a thread-private descriptor to direct the static call to use the thread-private TLT to transfer control to the thread-private code cache.

Specifically, the STM runtime provides a runtime library *stmGetTxnDesc()* to get a thread-private transaction descriptor. All the thread-private memory accesses go through the thread-private transaction descriptor. We add one field in the transaction descriptor, which points to the thread-private TLT. To access the thread-private TLT, STM compiler generates a global index variable storing an index to the thread-private TLT, for each call site to a legacy function. For program with separately compiled modules, our compiler puts all the index variables into a special segment *.tlt* in the object file and the program linker will combine them together into a global index table (GIT). At the program initialization for STM, we initialize each entry in the GIT with the TLT table offset. When a thread is initialized, we allocate and initialize a thread-private TLT for the thread with the same size as the GIT. Then we can save the thread-private TLT pointer in the transaction descriptor and use the global index variable to access the entry in the thread-private TLT for a particular function call. Fig. 3 shows the code for program initialization and thread initialization. *GIT_head* is the first index variable in GIT and *GIT_tail* is the last index variable in GIT. The

```
// program initialization
prog_init() {
  ...
  Index = 0;
  for(p = &GIT_head;
    p != &GIT_tail; p++) {
    *p = index++;
  }
  GIT_size = index;
}
```

(a) program initialization

```
// thread initialization
thread_init() {
  ... // allocate transaction descriptor
  desc = stmGetTxnDesc();
  desc->TLT = malloc(GIT_size * ENTRY_SIZE);
  For(index = 0; index < GIT_size; Index++) {
    if (the call site is a direct call)
        desc->TLT[index].proc = LDBTOM;
    else
        desc->TLT[index].proc = dispatch_lookup;
  }
}
```

(b) thread initialization

Fig. 3. Initialization for thread-private TLT

STM runtime defines these two index variables and makes sure they are the first entry and last entry in the GIT. *GIT_size* is the GIT table size.

Fig. 4 shows the code for a function call to legacy function in transaction. index_L is the index variable for function call at L. We use the thread-private descriptor and index_L to access the thread-private TLT entry and make an indirect function call to LDBTOM. LDBTOM can easily patch the thread-private TLT entry with the translated code in the thread-private code cache so that future function calls go directly to the translated code.

```
int foo (void);

main (…){
   #pragma tm_atomic
   {
      …
      L: foo();
   }
}
```

(a) source code

```
int  index_L; // in segment .tlt

main (…) {
   #pragma tm_atomic
   {
      …
      desc = stmGetTxnDesc();
      …
      call                    (*desc-
>TLT[index_L].proc)(…);
   }
}
```

(b) STM code

Fig. 4. Thread-private TLT for legacy function call

Another issue is determining the target of a function pointer and generating correct code when the function pointer is called inside a transaction. A function pointer always points to the normal version of a function if it is compiled by the STM compiler, or a legacy binary function. If it points to an STM compiler generated function, the transactional version must be called instead. If it points to a legacy binary function, however, the LDBTOM must be invoked to translate the function to transactional version.

In the source code, a function pointer can be declared as transactional with a tm_function pragma, such as fp in Fig. 5(a). The tm_function information can be

maintained by the compiler as part of the type of the function declaration. However, for an unmanaged language like C, the compiler seldom performs strict type checking, especially when the program modules are compiled separately. For example, if the code in Fig. 5 (a) and (b) are separately compiled and then linked together, the program will compile fine without any error or warning, although the call through fp() actually goes to a binary function (bar), even though it is declared as a transactional function pointer.

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

#pragma tm_function
int (fp *) (void);

main (…) {
    func2();
    #pragma tm_atomic
    {
        …
        foo();
        fp();
    }
}
```

(a) function call

```
  extern    int    (fp    *)
(void);
  int bar(void) { … }

  func2 (){
     …
     fp = bar;
     …
  }
```

(b) function declaration

Fig. 5. Transactional function pointers

```
// declared to be atomic
#pragma tm_function
int foo(int)
…
fp1 = foo;
fp2 = qsort; // lib

#pragma tm_atomic
{
  (*fp1)(3);
  (*fp2)(…)
}
```

(a) source code

```
#pragma tm_atomic
{
if (*fp1== "no-op marker")
  call **( fp1 – 4)
else
  Invoke_LDBTOM (fp1);
if (*fp2== "no-op marker")
  call **( fp2 – 4)
else
  Invoke_LDBTOM (fp2);
}
```

(b) runtime checking code

```
<foo-4>:
# address of
# transactional clone
 foo_tm
<foo>:
# no-op marker
  cmpl %eax, 0xmagic
# actual function here
<foo_1>:
 push %ebp;
 …
```

(c) assembly code supporting runtime check

Fig. 6. Handling function pointers inside transactions

Our STM compiler inserts a special "no-op marker" in the beginning of the normal version of the function it generated. It also places the address of the transactional clone at a fixed offset, e.g. 4 bytes, away from the normal function entry. The "no-op marker" is a special no-op unique to the user application being compiled, such as an instruction that loads a special STM reserved memory to an unused register. Since a legacy library function does not have this special marker, this allows a quick runtime check to determine whether a function is an STM compiler generated function or a legacy library function. For a call through a function pointer invoked inside a

transaction, if the pointer points to a STM compiler generated normal function, the pointer is adjusted and the transactional clone is called. If the pointer points to a legacy library function, LDBTOM is invoked to translate the code in legacy library.

Fig. 6 shows the indirect calls with function pointers. The transaction first calls the function *foo()* through the function pointer *fp1*. The code generated for the call to *foo()* inside the transaction first checks that the first instruction at the function entry is the no-op marker, and then calls the transactional clone of *foo()* pointed to by *(fp1 – 4)*. The transaction next calls the function *qsort* through the function pointer *fp2*. Since *qsort* is a legacy function whose first instruction is not the no-op marker, LDBTOM is invoked to translate the *qsort* function to a transactional clone. 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 no-op marker instruction.

5 Dynamic Optimizations

For each memory reference in a legacy function called in a transaction, LDBTOM inserts code to save the machine context (e.g. registers), pass the memory address to the STM runtime, call an STM runtime routine (*stmReadBarrier* for load and *stmWriteBarrier* for write) to check for conflict, and then restore the machine context. Fig. 7 shows the basic instrumentation for one memory reference. The basic instrumentation incurs noticeable overhead if not optimized.

movl %eax, DWORD PTR [%ebp+08h]	*save %eflags* *save caller saved regs* *pass PTR [%ebp+08h] to STM runtime* *call stmReadBarrier* *addl %esp, 0x4h* *restore caller saved regs* *restore %eflags* *movl %eax, DWORD PTR [%ebp+08h]*
(a) before translation	**(b) after translation**

Fig. 7. Basic Instrumentation for Legacy code

LDBTOM performs the following optimizations to reduce the barrier overhead.

- Analyze the binary code to identify thread-private memory references, such as push and pop operations, which do not need barriers.
- Use liveness information to eliminate unnecessary register save/restore. For example, if a register or the *%eflags* is dead at a reference, it does not need to be saved and restored for that reference. For another example, if a register is not used in a block/trace, it does not need to be saved or restored for each instrumentation site in the block/trace, and saving/restoring the register at the block/trace boundary is sufficient.
- Inline the STM runtime call to eliminate the call and return overhead. Since aggressive inlining increases code size dramatically, we inline only the hot paths in the STM runtime routines.

5.1 Filtering Local References

IA32 program frequently uses stack memory to store local data due to the limited number of general purpose registers. One benefit of stack data is that it is often private to a thread, and we may not need to track them for conflict detection. However, there are cases where the address of a stack data is passed to another thread and thus the stack data become shared among threads, although this situation happens very rarely. To filter barriers for stack references safely and aggressively, we need to solve the following two problems.

1. Decide whether any local data on stack is shared among different threads.
2. Decide which memory operation is a stack local data access.

Neither of the above problems can be easily solved by statically scanning the binary code. We solve the problems by taking the advantage of the dynamic binary translation opportunity. To check that no stack data are shared among threads, we leverage the paging memory protection mechanism provided in IA32 ISA. Each thread maintains its own paging table and protects the stack area as accessible only by the owner thread. In case that a thread accesses the stack of other threads, a memory fault triggers LDBTOM to flush all the translated code in the code cache and retranslates them without further filtering optimization of the barriers for stack references.

Stack references often are indexed by the stack pointer register *%esp* and the stack frame register *%ebp*. To check that *%esp* register points to the thread stack, LDBTOM examines all instructions that update *%esp* register with a large constant or a variable (With this update, the program may switch stack to memory space unprotected by our memory protection mechanism). If such an update is detected, LDBTOM flushes the code cache and retranslates the program without further filtering optimization.

The %ebp registers may be temporarily used as scratch register in leaf functions. We use the dynamic control flow graph information to track the *%ebp* status. An assignment that moves *%esp* to *%ebp* will cause *%ebp* pointing to the stack. If all the predecessors of a block being translated have *%ebp* pointing to the stack, the current block can performs the filtering optimization. If an instructions updates *%ebp* register with a large constant or a variable, the stack filtering optimization will not be performed. In case LDBTOM finds out that *%ebp* does not point to the stack at the end of a newly translated block, but a successor block has already been optimized based on the assumption that *%ebp* points to the stack, LDBTOM will create a new version of the successor block to run without filtering optimization. LDBTOM also dynamically checks *%ebp* after a function call returns. If *%ebp* no longer points to stack after the function call returns, code cache will be flushed and retranslated.

5.2 Dead Saving/Restore Elimination

In the base implementation LDBTOM saves/restores all caller-saved registers (*%eax*, *%ecx* and *%edx*) and *%eflags* register at each barrier. This is not always necessary as some registers may be dead at the barrier site. We use a simple analysis to detect the liveness of registers, and remove those unnecessary register saves/restores.

To eliminate dead saving/restores, each unfiltered load/store instruction maintains a local status word for *%eax*, *%ecx*, *%edx*, and *%eflags*. A global status word is initially set with "unknown" for all the registers. During a reverse traversal of the instructions, when a register is defined as a destination register, it is set as "dead" in the global status word. If a register is used as source register, it is set as "live". After an unfiltered load/store instruction is processed, the global status is assigned to the local status word of the load/store instruction. After all instructions in the block are scanned, if a register is never used in the block (with an "unknown" status), we save the register at the beginning and restore it at end of the block. For the other registers, we don't need to save/restore them at an unfiltered load/store if the register is dead in the local status word.

5.3 Barrier Inlining

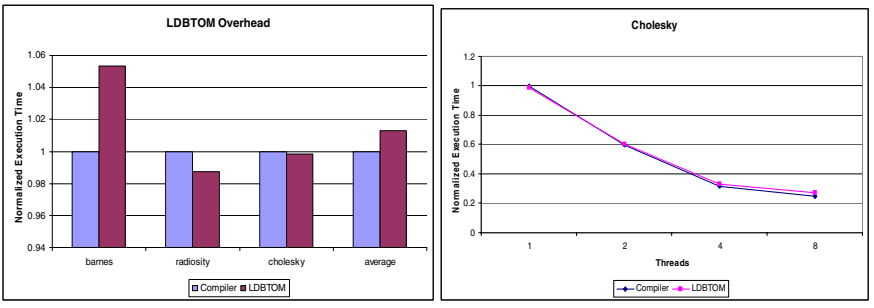
Inlining the barrier routines can reduce function call and return overhead. But inlining the entire barrier increases code size significantly, so we only inline hot paths, and switch to slow path when conflict detected or READ/WRITE set buffer is full. The hot paths are identified the same as in the STM compiler [7].

6 Experimental Results

We use a suite of SPLASH-2 benchmark and three concurrent data structure benchmarks, *avltree*, *btree* and *hashtable*, to demonstrate LDBTOM support in the STM compiler. We run our experiment on a Unisys ES7000 Linux system, with 16 processors. Each processor is an Intel Xeon MP CPU 3.0G Hz, with 8KB L1 Data cache, 512K L2 Cache, 4M L3 Cache, and 32M on-board L4 Cache. The system has a 400MHz system bus, 3.2GB/s front side bus bandwidth, 8G main memory, and a dual channel DDR 400. Each data point in the result is obtained from the average of 10 runs.

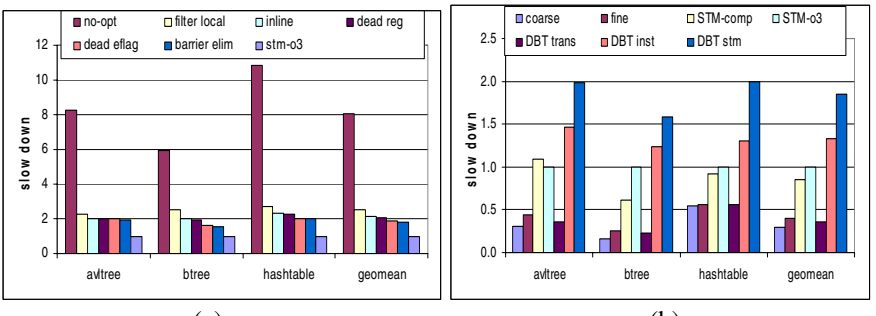
In real-world applications like SPLASH-2, only a small portion of code (on average 4%) runs in transactions. Among them, three benchmarks, *barnes*, *radiosity* and *cholesky*, have function calls within transactions, but none of them are external library calls. To measure the overhead of LDBTOM, we compile the program in two different ways: 1) declare all the functions called insides transactions as *tm_function* so that the compiler will generate transactional code for these functions (denoted as *Compiler* in the figures), and 2) do not declare any function called inside transaction as *tm_function* so that the compiler will invoke LDBTOM to dynamically translate them into transactional code (denoted as *LDBTOM* in the figures). Fig. 8 (a) shows the performance results running in a single thread. On average, the overhead of LDBTOM is only about 1%.

Fig. 8 (b) shows the scalability for the benchmark *Cholesky*. Since the LDBTOM overhead is very small, the scalability is similar for both versions. The other benchmarks also show similar scalability between Compiler version and LDBTOM version.



(a) (b)

Fig. 8. LDBTOM for SPLASH2



(a) (b)

Fig. 9. Optimization benefits and translation overhead

Since the SPLASH-2 has only a small portion of code running in transactions, we also use a set of concurrent data structures benchmarks, which spend almost all their execution time inside transactions to stress the LDBTOM overhead. For the concurrent data structures, each benchmark has hand-coded version optimized for STM execution, together with a fine-grain locking version and a coarse-grain transaction version. For the transaction version of the benchmarks, each coarse-grain locked section is coded as a transaction. Most of the transactions in the programs consist of a number of function calls. To stress LDBTOM overhead, we treat all the functions called inside the transactions as library functions translated by LDBTOM. We measure the performance of LDBTOM, using the hand-optimized version (**stm-o3**) as the baseline, with no optimization (**no-opt**) and with the following optimizations, running on a single processor.

filtering local:	Filter local variables indexed by esp/ebp
inlining:	Inlining fast path + Filtering local
dead reg:	Eliminate dead register saving/restore+ Inlining
dead eflag:	Eliminate dead *%eflags* and reg saving /restore
barrier elim:	Redundant barrier eliminations+dead *%eflags*

Fig. 9 (a) shows the optimization benefits. On the average, without optimization LDBTOM translated code is about 8x slower than hand-coded version. After optimizations, LDBTOM translated code is only about 80% slower than hand-coded version. The most beneficial optimizations are local filtering. All other optimizations contribute to the performance improvement, but not as significantly.

We also classify the overhead in LDBTOM translated code. We compare the performance for the following cases running with a single thread.

fine: this is the original fine-grain locking version of the benchmarks.

coarse: this is the original coarse-grain locking version of the benchmarks.

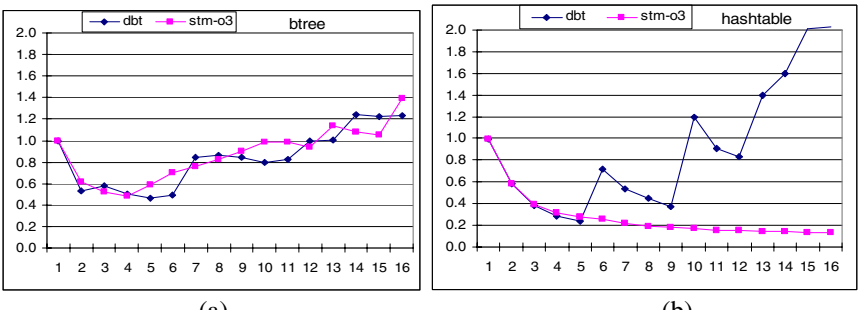
STM-o3: this is the hand-coded STM version.

STM-comp: this is the version generated by our STM compiler. This version is highly optimized and achieves performance similar to or better than the hand-optimized versions.

DBT trans: this is the transaction version with the functions translated by LDBTOM, but no STM instrumentation is inserted. This version demonstrates the overhead of invoking DBT to translate the function. This version can only run with a single thread, as the load/stores are not instrumented for conflict detection

DBT inst: this is the same version as *DBT trans* but also with the saving/restoring of registers code inserted. The actual calls to STM runtime routines are not inserted. This version measures the overhead associated with saving/restoring of machine context. This version can only run with a single thread.

DBT stm: this is the same version as *DBT inst* but also with the actual barrier code for conflict detection being inserted.



(a) (b)

Fig. 10. Scalability of benchmark

Fig. 9 (b) shows that LDBTOM translation overhead is small (compare bars marked with **DBT trans** and **fine**). This overhead can be even smaller for larger applications than the data structures, since these benchmarks run only for a short time and the DBT startup time is more noticeable. The overhead to save/restore machine context is relatively high, and **DBT inst** has about 3X higher overhead than **DBT trans**. When the STM barriers are inserted, the DBT translated version (**DBT stm**) is about 80% slower than the hand-optimized STM code (**STM-o3**).

We also compare the scalability of the code translated by LDBTOM with the hand-coded version. Fig. 10 (a) shows that LDBTOM generated *btree* scales similar to the hand-optimized STM code. For *hashtable*, however, Fig. 10 (b) shows that the LDBTOM generated code scales only to five processors. With 6 or more processors, the LDBTOM generated code for *hashtable* actually runs slower with more cores. This is because that currently LDBTOM uses thread-private code cache to store translated code. Consequently, for a parallel program running with multiple, say 16, threads, the legacy code in the parallel regions will be translated 16 times, one for each thread. Since translation for different threads cannot fully be performed in parallel yet, the translation overhead will increase when running with more threads. There is also potential issue with the increased pressure on the instruction cache. We plan to address this issue in two directions in the future: 1) make the LDBTOM translation and optimization for different threads fully parallel, and 2) implement a shared code cache so the translated code can be reused by multiple threads. The zigzag in Fig. 10 (b) also suggests that there are load imbalance issues.

7 Summary and Future Work

In this paper, we develop novel techniques for integrating a dynamic binary translation module into a STM compilation environment to support transactional memory for legacy binary code, and evaluate a number of optimization techniques to reduce the overhead of the translated code. We measure the effectiveness of these techniques on a suite of SPLASH-2 benchmarks and a set of concurrent data structures benchmark. For the SPLASH-2 benchmarks, on the average, the LDBTOM generated code is only about 1% slower than STM compiler generated code. For the concurrent data structure benchmarks which have almost all the code inside transactions, LDBTOM generated code is about 80% slower than the hand-optimized STM code on a single thread, even though binary code is inherently harder to optimize than high-level source code and a straightforward translation would be more than 8 times slower than the hand-optimized code.

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 signal handling inside a transaction. Software implemented synchronizations using shared variables pose a challenge to the dynamic binary translator to convert them to transactional code. The scalability issue with thread-private cache is also an interesting future research topic.

References

1. Adl-Tabatabai, A., Lewis, B.T., Menon, V.S., Murphy, B.M., Saha, B.: T. Compiler and runtime support for efficient software transactional memory. In: PLDI 2006 (2006)
2. Adl-Tabatabai, et al.: Hw Acceleration For a Software Transactional Memory System. In: Micro 2006 (2006)
3. Ananian, C.S., Asanovic, K., Kuszmaul, B.C., Leiserson, C.E., Lie, S.: Unbounded transactional memory. In: 11th International Symposium on High-Performance Computer Architecture, 2005. HPCA-11, February 12-16, 2005, pp. 316–327 (2005)

4. Baraz, L., Devor, T., Etzion, O., Goldenberg, S., Skaletsky, A., Wang, Y., Zemach, Y.: IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems. In: Micro-36 2003 (2003)
5. Blundell, C., Lewis, E.C., Martin, M.M.K.: Deconstructing Transactional Semantics: The Subtleties of Atomicity. In: Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD) (June 2005)
6. Blundell, C., Lewis, E.C., Martin, M.M.K.: Unrestricted Transactional Memory: Supporting I/O and System Calls within Transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA (April 2006)
7. Wang, C., Chen, W., Wu, Y., Saha, B., Adl-babatabai, A.: Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In: CGO 2007 (2007)
8. Wang, C., Hu, S., Kim, H.-s., Nair, S., Breternitz Jr., M., Ying, Z., Wu, Y.: StarDBT: An Efficient Multi-platform Dynamic Binary Translation System. In: Choi, L., Paek, Y., Cho, S. (eds.) ACSAC 2007. LNCS, vol. 4697, pp. 4–15. Springer, Heidelberg (2007)
9. Borin, E., Wang, C., Wu, Y., Araujo, G.: Software-Based Transparent and Comprehensive Control-Flow Error Detection. In: CGO 2006 (2006)
10. Qin, F., Wang, C., Li, Z., Kim, H.-s., Zhou, Y., Wu, Y.: LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In: Micro-39 2006 (2006)
11. Harris, T., Plesko, M., Shinnar, A., Tarditi, D.: Optimizing Memory Transactions. In: PLDI 2006 (2006)
12. Herlihy, M., Luchango, V., Moir, M., Scherer, W.N.: Software Transactional Memory for Dynamic Sized Data Structures. In: PODC 2003 (2003)
13. Hosking, A., Moss, J.E.B.: Nested transactional memory: Model and preliminary Sketches SCOOL (2005)
14. Olszewski, M., Cutler, J., Steffan, J.G.: JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In: PACT 2007 (2007)
15. Moir, M.: Hybrid Transactional Memory. Sun Microsystems Technical Report
16. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A.: LogTM: Log-based Transactional Memory. In: HPCA 2006 (2006)
17. Wu, Q., Reddi, V.J., Wu, Y., Lee, J., Connors, D., Brooks, D., Martonosi, M., Clark, D.W.: Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance. In: Micro-38 2005(2005)
18. Rajwar, R., Herlihy, M., Lai, K.: Virtualizing Transactional Memory. In: Proc. of the 32nd Annual Intl. Symp. On Computer Architecture (June 2005)
19. Shavit, N., Tuitou, D.: Software Transactional Memory. In: PODC 1995(1995)
20. Correct and Consistent Transactional Memory System (submitted for publication)