Transactifying Applications using an Open Compiler Framework

Pascal Felber
University of Neuchâtel, Switzerland
pascal.felber@unine.ch

Christof Fetzer
TU Dresden, Germany
christof.fetzer@tu-dresden.de

Ulrich Müller
TU Dresden, Germany
um@se.inf.tu-dresden.de

Torvald Riegel
TU Dresden, Germany
torvald.riegel@tu-dresden.de

Martin Süßkraut
TU Dresden, Germany
martin.suesskraut@tu-dresden.de

Heiko Sturzrehm *
University of Neuchâtel, Switzerland
heiko.sturzrehm@unine.ch

Abstract

Transactional memory dramatically reduces the complexity of writing concurrent code. Yet, seamless integration of transactional constructs in application code typically comes with a significant performance penalty. Recent studies have shown that compiler support allows producing highly efficient STM-based applications without putting the hassle on the programmer. So far, STM integration has been partially implemented in custom, proprietary compiler infrastructures. In this paper, we propose and evaluate the use of the LLVM open compiler framework to generate efficient concurrent applications using word-based STM libraries. Since LLVM uses the GCC compiler suite as front-end, it can process code written in C or C++ (with partial support for other languages). We also present a tool that allows "transactifying" assembly code and can complement LLVM for legacy code and libraries. Experiments using a lightweight C word-based STM library show that LLVM integration performs as well as hand-optimized calls to the STM library and better than assembly code instrumentation of the application code.

1. Introduction

Writing concurrent application using transactional memory is a huge leap forward from traditional lock-based synchronization. Yet, developers usually have two choices: either they program explicit calls to the transactional memory; or they opt for semi-transparent transactional support, i.e., they leave it to the system to "transactify" the application. The latter can be based on the programmer's use of dedicated language constructs [4] or on declarative mechanisms and AOP code weaving [11].

* This research was partially supported by the by the Swiss National Science Foundation under grant number 5005-67322 (NCCR-MICS).

Using explicit calls to the transactional memory, developers are exposed to the relative complexity of transaction management but they can optimize the code for performance. For example, the developer might know that some variables are not shared and hence, do not need to be accessed through the transactional memory. With semi-transparent transactional support, the price to pay for ease of programming is usually a reduced efficiency. It is not trivial, for instance, to avoid redundant accesses to transactional memory or to identify read-only transactions *a priori*.

Compiler and runtime support for transactional memory can in many cases alleviate these limitations and produce highly efficient code without putting much burden on the developer. Indeed, one can expect that a compiler can identify transactional memory accesses with relative ease and map them to the underlying STM. Furthermore, one can take advantage of the optimization passes of the compiler to reduce as much as possible the costly operations on transactional memory, and hence improve the performance of the resulting code.

For many researchers in the field, coming up with good compiler support for STM is just not feasible. Existing compiler work in the STM domain is (as far as we know) based on proprietary compilers. However, without compiler support, it is difficult to get good and representative workloads and it is very difficult to get comparable performance results, i.e., research progress will be slower. As a result for the community, some good ideas might be lost and smaller research groups will find it increasingly hard to contribute.

Modifying an existing compiler like `gcc` is technically difficult and it would be politically difficult to get extensions like transactional memory support into the main branch of the compiler. However, such optimizing tools are needed to convince developers of the benefits of STM and trigger the demand. Instead of modifying an existing monolithic compiler, we are investigating how far reusing the components

of an existing compiler framework (which provides parts such as front-ends, back-ends for different platforms, or link time optimizers) would be sufficient to optimize transactional programs. To do so, we transactify programs by instrumenting a machine-independent intermediate representation or machine specific assembly code. This is technically easier and the same framework can be used in combination with multiple front-ends supporting a variety of programming languages.

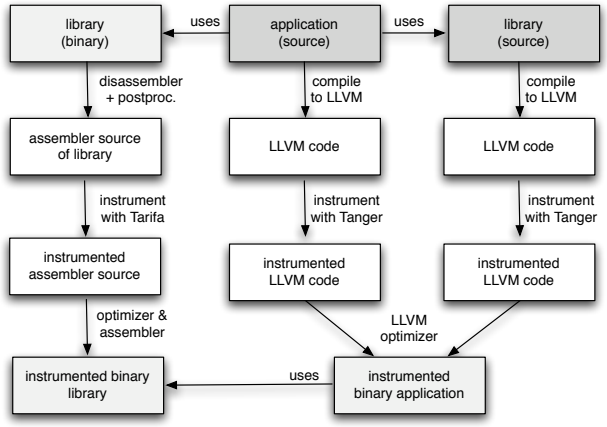

Figure 1. Architecture of our two tools. TARIFA instruments machine-specific x86 assembly code while TANGER instruments machine-independent LLVM code.

We have implemented two instrumentation tools to semi-transparently transactify programs written in unmanaged languages like C and C++ (see Figure 1). The first tool, TANGER, statically instruments intermediate code produced by the LLVM [8] compilation framework. LLVM has not only gained popularity in the academic domain but it is also expected to be used in the upcoming release of OS X version 10.5.

LLVM has important advantages in that it facilitates a hardware-independent intermediate representation (IR) for code. Application code in the LLVM IR can be easily modified by LLVM components such as compiler passes. LLVM IR can be seamlessly transformed between in-memory and on-disk representations and a human-readable assembler language. Different compilation phases are accessible as separate tools, so new transformations such as STM instrumentation can be provided without having to modify existing LLVM components. This eases adding STM instrumentation to existing build procedures (LLVM provides a `gcc`-like interface), as well as making deployment of STM instrumentation tools easier (i.e., in constrast to monolithic compilers, deploying a modified compiler is not necessary). LLVM additionally provides link-time optimizations, just-in-time compilation, compilation front-ends (e.g., a `gcc`-based front-end for C/C++), and code generation back-ends for various architectures as well as for generating C source code from LLVM IR.

To understand the need for a second tool, consider that application programs will sometimes have to call library functions from within atomic blocks. The problem is that not all libraries are available as source code, and even for those for which one has access to the source code, it is often not known how it was initially configured and compiled. Hence, it is difficult to recompile and transactify these libraries automatically. Therefore, our second tool, TARIFA, statically instruments x86 assembly code, so that we can transactify existing library code that is called from within atomic blocks.

The decoupling of the instrumentation via TANGER and TARIFA from the compiler and its optimization components permits the use of an unmodified compiler and optimizer. In particular, we can always use the latest version of the compiler and optimizer without the need to adapt changes for every new release (as needed in other approaches). We do not yet provide optimizations specially targeted as STM instrumentation. However, we have observed that transactional code already benefits from the general-purpose optimizations that LLVM performs. For example, redundant loads from memory are transformed into a single load into a virtual register, which reduces the number of transactional loads introduced by our instrumentation. Since it is difficult to compare with other existing compile-time approaches, we compare our approach with manually instrumented and optimized code compiled via LLVM, as well as with standard `gcc`. We will see that the LLVM code is typically as good as the manually optimized code in our benchmarks.

Related Work

Compiler support for transactional memory was first used in languages with managed environments. Harris and Fraser first employed compiler support to provide atomic blocks in Java programs [4]. Adl-Tabatabai *et al.* describe support for transactions in an optimizing compiler in [1]. They provide a managed runtime environment that supports multi-threaded Java programs using transactional language extensions. Their platform is based on a just-in-time dynamic compiler for Java and C#, a virtual machine, and the McRT-STM runtime [13]. Harris *et al.* present how to decrease runtime overheads by optimizating for transaction support in an compiler for Common Intermediate Language programs [5]. Transaction support based on Java byte code modification is presented by Herlihy *et al.* in [6]. We use a similar approach in newer versions of LSA-STM [11].

Transaction support for unmanaged languages is described in [17], and several optimizations for a C compiler are presented (e.g., moving transactional loads out of loops). Damron *et al.* also use a modified C compiler to support transactions [2].

There exist a variety of binary instrumentation tools. The first instrumentation tool was (as far as we know) ATOM [16] which is a static binary instrumentation tool.

ATOM permits to iterate over all functions in a binary and over all instructions within a function and permits to add new instructions before and after existing instructions. ATOM requires changes to the tool chain (i.e., compiler and linker) to make sure one can implement the iterators. Static instrumentation has the disadvantage that one either needs changes to the tool chain or one cannot guarantee that one can always distinguish between what is data and what is code. However, one can cope with these problems using several heuristics and in particular, one can flag parts of the code that might contain disassembling errors [15].

Most modern instrumentation tools are dynamic, i.e., they instrument basic blocks at the time they are executed the first time. There are a variety of dynamic binary tools available, e.g., PIN [9] and Valgrind [10]. Typically, dynamic instrumentation tools exhibit a large run-time overhead and hence, are not ideal for transactifying programs and libraries.

Dynamic binary translation and optimization is used to transactify binary legacy code in [18]. Overheads are reduced by applying optimizations such as removing transactional accesses to thread-private data, inlining, or avoiding saving and restoring dead registers and processor flags.

Link-time optimizers like PLTO [14] permit the static optimizations of binaries. One could use appropriate binary optimizers to optimize the code generated by our TARIFA tool. Also, a link-time optimizer needs to perform some form of static binary instrumentation itself and might hence be used to instrument binary code. However, we decided to write our own static instrumentation tool for x86 (TARIFA) because we need a very specialized instrumentation targeted towards individual address modes instead of individual operations.

Contributions

Most of the current research focuses on modern languages like Java and C# that use intermediate codes and just-in-time compilation. This work focuses instead on supporting transactional memory in C and C++. Both languages are known for their efficiency and hence they are used as the primary language in a large percentage of applications. However, to the best of our knowledge, there is so far no transparent and widely available STM support for C and C++ applications. Before deciding to use LLVM for instrumention, we have experimented with aspect-oriented extensions for C and C++. However, the AOP tools for C and C++ were neither sufficiently stable nor was the resulting code sufficiently fast.

In this paper, we evaluate the use of an *open* compiler framework as a generic front-end to word-based STM implementations. Our goal is to provide the means to develop efficient STM-based applications without putting much hassle on the developer and allowing designers of STM infrastructures to plug in their own STM implementations. The resulting tool, TANGER, relies on the aggressive optimization of the link-time optimizer to produce applications that perform competitively with hand-optimized code

Using an open compiler framework has the advantages that it not only provides a means to the community to share instrumentation tools and workloads, but also to build common benchmarks in which new STMs and optimizers can be plugged in. This will hopefully result in better comparisons of the proposed optimizers and STM implementations.

We also present TARIFA, a tool that allows us to instrument x86 assembly code and map memory accesses to the underlying transactional memory. Although TARIFA does not produce code that is as efficient as the instrumented LLVM code, it can be used to transactify legacy code or libraries that must be used from within transactions (a transaction should only call functions that are themselves transactional). Therefore, the combination of TANGER and TARIFA can support sophisticated applications with dependencies on legacy code.

Finally, we have also developed a low-footprint, efficient word-based STM library written in C to evaluate TANGER and TARIFA. This library, called TINYSTM, is the basis for our development of tools for supporting resource-limited embedded applications that use STMs for parallelization and failure atomicity.

Our preliminary performance evaluation reveals that TANGER produces code that is as efficient as hand-optimized code, while TARIFA does not perform as well due to the complexity of identifing and optimizing transactional memory operations at the assembly code level.

2. TANGER: STM Support with LLVM

LLVM [8] is a modular compiler infrastructure that provides language- and target-independent components. It defines an intermediate representation (IR) for programs with a simple type system. In this way, LLVM provides a clean way for instrumenting programs and adding calls to an STM. The instrumented code can then be optimized and a machine-specific binary is statically generated from the optimized bytecode.

LLVM simplifies the instrumentation dramatically in comparison to instrumenting native x86 code (see Section 3) because the LLVM IR is based on a *load/store architecture*, i.e., programs transfer values between registers and memory solely via load and store operations using typed pointers. This permits us to translate memory accesses to accesses to transactional memory in a straightforward way.

LLVM IR has no concept of a stack. Local variables are not stored on the stack and are thus not accessed via loads and stores. Instead, all local variables are kept in registers. Exceptions are local variables that are accessed via pointers: they are explicitly allocated, their addresses are kept in registers, and they are accessed via explicit load and stores. Hence, many local variables are not accessed via the STM because we do not instrument register accesses. This can decrease the runtime overhead of instrumented code and works without any additional glue code as long as transactions

cover the whole body of a function (i.e., transactions have function granularity). If transactions only cover a subset of a function, the instrumentation needs to take care of the rollback of local variables before a retry or an abort of a transaction. This rollback is simplified because LLVM uses a single static assignment form.

The LLVM framework has the possibility to alter the intermediate representation (IR) through so-called "passes". These passes, written in C++, can iterate at different granularities over the IR. Our TANGER instrumentation tool is implemented as an LLVM pass. To transactify an application, we first use LLVM to compile to the IR and apply general-purpose optimization passes, then transactify the IR using TANGER, and finally apply optimizations again and use LLVM backends to produce native binary code or C source code.

Unlike memory accesses that are implicitely translated to transactional accesses with no intervention of the programmer, transaction demarcation must still be specified explicitely. An exception is when transactions cover the whole body of a function and can be specified declaratively (by providing a list of "transactional functions" to TANGER).

```
1  int set_contains ( intset_t *set, int val)
2  {
3    int result;
4    node_t *prev, *next;
5
6    atomic(
7      prev = set->head;
8      next = prev->next;
9      while (1) {
10       v = next->val;
11       if (v >= val)
12         break;
13       prev = next;
14       next = prev->next;
15     }
16     result = (v == val);
17   )
18
19   return result;
20 }
```

Figure 2. Original C code with atomic block for testing containment in an integer set.

To specify transaction demarcation around arbitrary regions of code, we can either use "atomic blocks" (using the `atomic` keyword, see Figure 2) and use a source code preprocessor, or surround transaction by macros like `ATOMIC_START` and `ATOMIC_END`. In both cases, the transaction start and end will be translated into calls to external functions that will appear in the LLVM bytecode and serve as markers for instrumentation.

After compiling the source code to LLVM byte code, the marker functions and all other load, store and call instructions are still in the same order as before (see Figure 3). Currently, we add a transaction descriptor as an additional argument to instrumented functions to avoid the extra cost of accessing a thread-local variable. Our tool, TANGER, replaces the start marker by a call to `setjmp` and one to `stm_start` (Figure 4, line 8–9). The `setjmp` call is needed for a later rollback via `longjmp` upon abort.[1] Between these two instructions, a label is inserted to have a jump point which is needed if the transaction needs to be retried. Each start marker can have several corresponding end markers calls due to branches and switches but they have to be at the same nesting level and in the same function. Using the atomic block syntax enforces that each start marker has one valid matching end marker. End markers are replaced by a `stm_commit` call and a branch which proceeds if the commit was successful or jumps to the start of the atomic block in case a retry is needed (lines 25–27).

```
1  int %set_contains(%struct.intset_t * %set, int %val) {
2  entry:
3    ...
4    %tmp = tail call %struct.stm_tx_t* (...)* %stm_get_tx( )
5    tail call void %startTANGER( %struct.stm_tx_t* %tmp )
6    %tmp22 = getelementptr %struct.intset_t * %set, int 0, uint 0
7    %tmp23 = load %struct.node_t** %tmp22
8    %tmp25 = getelementptr %struct.node_t* %tmp23, int 0, uint 1
9    %next.2 = load %struct.node_t** %tmp25
10   %tmp29 = getelementptr %struct.node_t* %next.2, int 0, uint 0
11   %tmp30 = load int* %tmp29
12   %tmp33 = setlt int %tmp30, %val
13   br bool %tmp33, label %cond_next, label %bb39
14   ...
15   tail call void %endTANGER( %struct.stm_tx_t* %tmp )
16   %result.0.in = seteq int %tmp30.1, %val
17   %result.0 = cast bool %result.0.in to int
18   ret int %result.0
19 }
```

Figure 3. LLVM bytecode generated from Figure 2.

Between the two transaction demarcation markers, TANGER has to transactify all memory accesses. As already mentioned, the instruction set of LLVM provides a very efficient way to identify such operations. There are only three instructions to deal with: load, store, and call. Because of the compilation via `llvm-gcc`, which uses the `gcc` frontend, occurence of these three instructions is already minimized through the optimization. Furthermore, most non-heap loads and stores are eliminated as well.

When a load or a store has been identified to be transactional, it is replaced by the corresponding call to the STM (`stm_load` or `stm_store`), possibly surrounded by cast instructions (lines 12, 16, and 20). Casting is necessary if the data type accessed by the program does not match one of the native types managed by the STM implementation. Such casts are necessary to preserve the type constraints of LLVM but typically they do not produce extra intructions in the binary (e.g., casting between pointer types as on line 11, between unsigned and signed type as on line 21, or between memory addresses and unsigned value of the same size as on line 13).

[1] If one does not want to use the `setjmp`/`longjmp` facility, one has to explicitely check whether a transaction aborted during its execution (e.g., after every load and store) and restart it if that is the case. The resulting code will be larger and slower.

```
...
cond_false :
 %tmp = tail call %struct.stm_tx_t* (...)* %stm_get_tx( )
 br label %cond_false.startTX

cond_false.startTX:
 %txref21 = getelementptr %struct.stm_tx_t* %tmp, int 0, uint 6, int 0
 %txref22 = tail call int %_setjmp( %struct.__jmp_buf_tag* %txref21 )
 tail call void %_stm_start( %struct.stm_tx_t* %tmp, int 0, int 1 )
 %tmp22 = getelementptr %struct.intset_t* %set, int 0, uint 0
 %castA23 = cast %struct.node_t** %tmp22 to uint*
 %callB23 = tail call uint %stm_load( %struct.stm_tx_t* %tmp, uint* %castA23 )
 %tmp231 = cast uint %callB23 to %struct.node_t*
 %tmp25 = getelementptr %struct.node_t* %tmp231, int 0, uint 1
 %castA24 = cast %struct.node_t** %tmp25 to uint*
 %callB24 = tail call uint %stm_load( %struct.stm_tx_t* %tmp, uint* %castA24 )
 %next.22 = cast uint %callB24 to %struct.node_t*
 %tmp29 = getelementptr %struct.node_t* %next.22, int 0, uint 0
 %castA25 = cast int* %tmp29 to uint*
 %callB25 = tail call uint %stm_load( %struct.stm_tx_t* %tmp, uint* %castA25 )
 %tmp303 = cast uint %callB25 to int
 %tmp33 = setlt int %tmp303, %val
 br bool %tmp33, label %cond_next, label %bb39
...
 %txref28 = tail call int %stm_commit( %struct.stm_tx_t* %tmp )
 %txref29 = seteq int %txref28, 0
 br bool %txref29, label %cond_false.startTX, label %bb39.commit30

bb39.commit30:
 ...
```

Figure 4. Instrumented LLVM bytecode generated from Figure 3.

Function calls need special handling. Some of them are calling external functions for which we cannot generate a transactional copy with TANGER. This has to be done, for instance, with TARIFA (see Section 3). For internal functions, our approach is to create a "transactional clone" of the function. If a function call is inside an atomic block or a transactional function, then the transactional clone is called. Otherwise, the regular function is called.

One should note that the instrumentation process of TANGER is configurable and can be easily adapted to generate different code for transactional operations. This would allow other groups to test their own STM implementations using our tool with little effort. If the interface of two STM implementations is the same, then switching to the other STM just requires recompilation. For example, we were able to compare our STM to TL2 [3].

TANGER currently only supports word-based STMs, but we believe that the LLVM infrastructure provides the necessary tools to support object-based STMs as well. The LLVM IR is typed; addresses are obtained by navigating through typed arrays and structures. For example, the following code shows a load of the first element of the `intset_t` structure pointed at by `set`:

```
%tmp = getelementptr %struct.intset_t* %set, int 0, uint 0
%tmp1 = load %struct.node_t** %tmp
```

Furthermore, LLVM provides alias analysis implementations, and whole-program analysis is also possible if all parts of the program are compiled using LLVM.

3. TARIFA: STM Support for Binary Code

TARIFA instruments native x86 code with calls to an underlying STM (see Figure 5). The primary goal of TARIFA is to be able to instrument existing libraries that export functions that are called from within atomic blocks. One can use a standard disassembler, e.g., `objdump`, to disassemble libraries. We process the output of the disassembler and use information from the dynamic link loader to transform the output into the right assembly format that can be processed by TARIFA. Disassembling binaries might be difficult because for a stripped binary it is not always possible to distinguish between code and data embedded in the code segment. However, one can check during the post-processing that a sufficient part of the library was disassembled, i.e., we have disassembled a slice of the code that contains all functions that might be called directly or indirectly by the application and during initialization of the library. Alternatively, we can also instrument assembly code generated by the compiler. We are currently supporting `gcc` assembler syntax, i.e., assembly files generated by `gcc` and `g++`. Note, however, that we recommend to generate LLVM code instead because this results in much more efficient code (see Section 4).

TARIFA transforms x86 assembly code such that memory access are transformed into calls to the underlying STM. Unlike LLVM, x86 is not a load and store architecture, i.e., memory locations might be accessed directly by operations like `add`. Hence, we need to transform all operations that access memory. Unlike also in LLVM, we do not have additional registers that we could easily use to load the arguments from the STM or to write intermediate results before writing to the STM. Hence, we need to push some registers temporarily on the stack. The resulting code might not be very efficient but it could easily be optimized, for example, to minimize the number of push and pop operations (see Figure 5). Ideally, we would like to use an existing link time optimizer like PLTO [14] to do this optimization.

4. Evaluation

To evaluate the efficiency of our TANGER and TARIFA tools, we have implemented a lightweight (less than 500 LOC) word-based STM in C called TINYSTM. It is a time-based transactional memory [12] that uses encounter-time locking, similar to [17], and guarantees consistent reads for every active transaction. TANGER, TARIFA, and TINYSTM are freely downloadable from `http://tinystm.org`.

For our preliminary performance evaluation, we have experimented with the "classical" *intset* micro-benchmark used in [7] and by several other groups. It consists of an integer set implemented as a sorted *linked list*.

The set was initially populated with 256 random elements (we also experimented with larger values and observed the same trends). Then, we started n threads performing concurrent read-only and update transactions (searching for a value and adding/removing an element). We kept the size of the

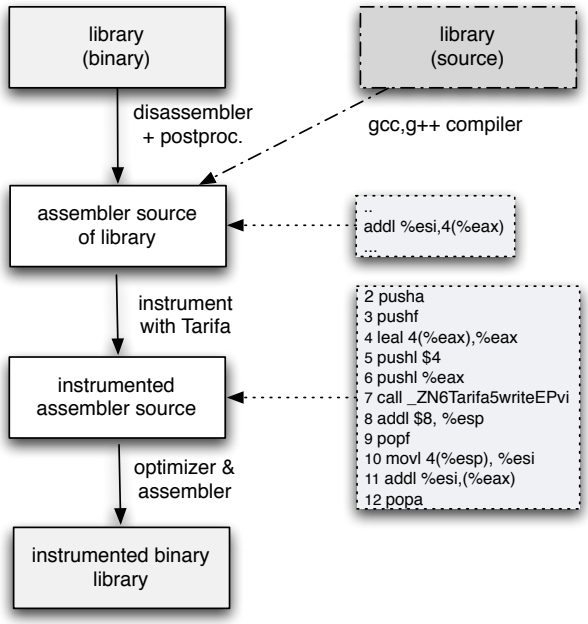

Figure 5. TARIFA instruments x86 assembly code generated by disassembling binary libraries or, alternatively, by the compiler.

integer set almost constant by alternatively adding and removing an element. The rate of update transactions was set to 20%. Each experiment was run five times and we kept the median value.

We have compared three version of the benchmark: (1) a version that was optimized by hand, i.e., we explicitely inserted the minimal number of transactional load and store operations; (2) a version that was instrumented using TANGER; and (3) a version that was instrumented using TARIFA. We compiled the first version, using both `gcc` and the LLVM compiler chain. Experiments were run on a two-way machine with dual-core AMD Opteron processors at 2GHz, which makes 4 cores.

Results are shown in Figure 6. One can observe that the version produced by TANGER performs as well as the hand-optimized version when scaling up the number of threads. The version instrumented using TARIFA runs slower due the additional accesses to the transactional memory (about twice as many), the management of the stack, and the suboptimal register allocation, but it still scales well. Interestingly, the LLVM compiler chain produced code that is as good as the code generated by `gcc`.

We have also evaluated the overhead of TARIFA in hybrid settings, where it is used for transactifying some parts of a transactional application. To that end, we have instrumented the removal operation of the integer set using TARIFA, and the other operations using explicit STM operations. We have

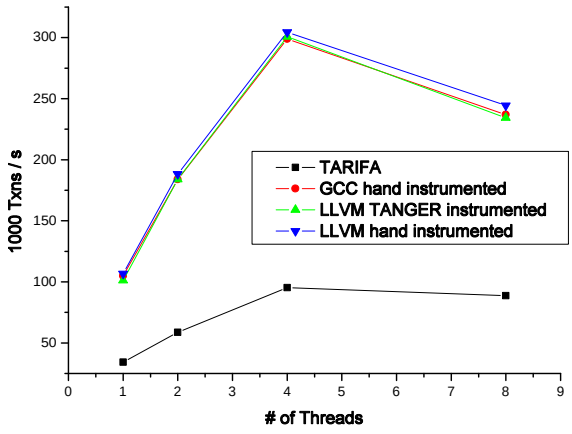

Figure 6. Performance of the TANGER, TARIFA, and hand-optimized benchmark versions.

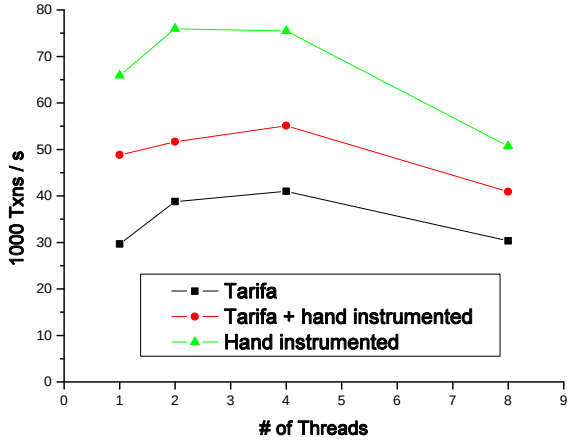

Figure 7. Performance in hybrid settings.

performed experiments under high contention with an update rate of 100%, i.e., we had only insertions and removals.

The results, depicted in Figure 7, show that the performance of the hybrid application is almost exactly in between that of the TARIFA and the `gcc` hand-optimized versions. This is not surprising given that half of the function calls are to TARIFA-instrumented functions and the other half to "native" STM functions.

Finally, we have tested TARIFA's ability at instrumenting binary libraries. We have selected a library containing a random number generator (RNG). The experiment was run in two setups: (1) each thread maintained a separate state for the RNG and (2) all threads shared one RNG state. The second setup demonstrates TARIFA's ability to transactify memory accesses within a binary library. We have decompiled the

library and sliced out all functions belonging to the RNG. The functions were then instrumented by TARIFA, and the library recompiled. We have modified the *intset* benchmark so that it uses the RNG of the instrumented library when adding or searching for an element. The benchmark was run with an update rate of 20%. Intuitively, the setup in which all threads share a common RNG state should create a sequential bottleneck because it introduces a globally shared variable that is updated at the beginning of each transaction when determining the element to look for, insert, or remove.[2]

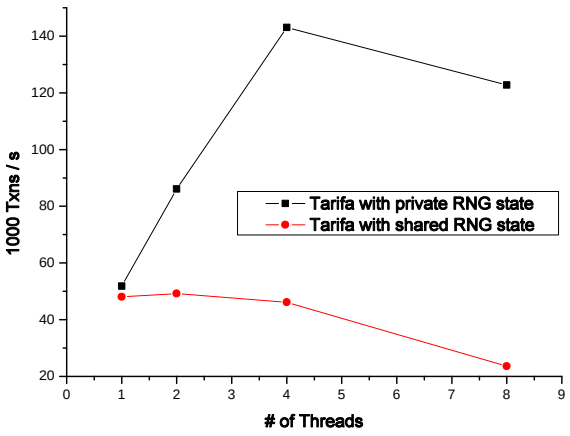

Figure 8. Demonstration of transactifying a binary library with and without sharded state using TARIFA.

We observe indeed in Figure 8 that, when each thread maintains a private RNG state, calls to the external RNG function do not result in conflicts and contention. However, if the RNG state is shared and as the number of threads grows, contention increases and the shared state becomes a sequential bottleneck. In contrast, performance scales with the number of threads if each thread maintains a private RNG state. This demonstrates that TARIFA transactified the accesses to the shared state of the binary RNG library.

5. Conclusion

We introduced two new tools to transactify programs: TANGER and TARIFA. TANGER instruments LLVM intermediate code, whereas TARIFA permits us to process x86 assembly code. Both tools can be used in collaboration to transactify programs that use external libraries for which one has no access to source code. Our goal is to support the parallelization of existing C and C++ applications by minimizing the effort required to transactify these programs.

Our secondary goal is to provide a framework for the STM community in which one can evaluate (1) new code optimizations in form of LLVM passes, (2) workloads, and (3) new STM designs and implementations. This might help to simplify the comparison of competing approaches. TANGER, TARIFA, and TINYSTM are available for download at `http://tinystm.org`.

[2] TINYSTM uses eager conflict detection for writes and does not allow more than one pending write per location.

References

[1] 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 *Proceedings of PLDI*, Jun 2006.

[2] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, New York, NY, USA, 2006. ACM Press.

[3] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *20th International Symposium on Distributed Computing (DISC)*, September 2006.

[4] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of OOPSLA*, pages 388–402, Oct 2003.

[5] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *Proceedings of PLDI*, Jun 2006.

[6] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 253–262, New York, NY, USA, 2006. ACM Press.

[7] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of PODC*, pages 92–101, Jul 2003.

[8] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[10] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, San Diego, California, USA, June 2007.

[11] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *TRANSACT06*, Jun 2006.

[12] T. Riegel, C. Fetzer, and P. Felber. Time-based Transactional Memory with Scalable Time Bases. In *19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, June 2007.

[13] 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 *Proceedings of PPoPP*, Jun 2006.

[14] B. Schwarz, S. Debray, and G. Andrews. Plto: A link-time optimizer for the intel ia-32 architecture. In *Workshop on Binary Translation (WBT)*, 2001.

[15] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *IEEE Working Conference on Reverse Engineering (WCRE)*, 2002.

[16] A. Srivastava. Atom: A system for building customized program analysis tools. In *SIGPLAN*, 1994.

[17] 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.

[18] V. Ying, C. Wang, Y. Wu, and X. Jiang. Dynamic Binary Translation and Optimization of Legacy Library Code in an STM Compilation Environment. In *WBIA*, 2006.