

# Providing Transparent Transactions on C/C++ Memory Locations with TARIFA

Martin Süßkraut      Ulrich Müller  
Dresden University of Technology  
Department of Computer Science  
Institute for System Architecture  
{ms67, um766083}@inf.tu-dresden.de

## 1 Introduction

The next micro processor generations will have multiple cores [3]. Programmers have to integrate parallelism in their applications to fully utilize the power of a multi-core processor. One has to consider race conditions, and dead locks when programming software with multiple threads. These issues compromise correctness and high assurance. To make the programming of multi-threaded software as simple as possible is one way to address them.

Today race conditions are handled by synchronizing access to shared data with mutual exclusion locks. But doing so can be very hard and there is always the risk to produce deadlocks or introduce unnecessary performance penalties. However, transactions can substitute locks in many situations [2] without typical locking problems because all transactions are executed atomically and run isolated from each other. Programmers put all access to shared data under the control of a transaction manager instead of using locks. If the contexts of two transactions collide one of them can be safely aborted and restarted after its changes are undone. Harris and Fraser showed [1] that a program with transactions can be much faster than a program with fine grained locking.

We propose *TARIFA* (short for TransActions by assem- bleR Instrumentation FrAamework) that transparently provides transactions to C/C++. We direct our attention towards integrating *Software Transactional Memory* (STM) implementations into the programming language by modifying all memory accesses on assembler level. A programmer doesn't need to find and modify all memory accesses by hand, which is an error-prone process.

## 2 TARIFA

Using an existing STM implementation in C/C++ that provides transaction functionality typically looks like this:

```
t = STMStartTransaction();
```

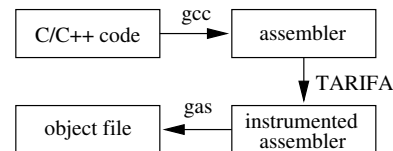


Figure 1. Work flow using TARIFA

```
STMWriteValue( t, &a, STMReadValue( t, &b) +  
              STMReadValue( t, &c) );  
STMCommitTransaction( t );
```

This function computes  $a = b + c$ ; guarded by a transaction. The above calculation looks non-intuitive. The use of the STM dominates the functional code. And a programmer will have to largely rewrite his code in order to use transactions.

The goal is to do the accesses to shared data transparently for the programmer. Extending a compiler like `gcc` would be too complex and not portable to other compilers. Our approach is to instrument the assembler output. *TARIFA* identifies all heap-memory read and write accesses in the assembler code that are part of a transaction and puts them under control of a STM.

Figure 1 shows the work flow. We compile the C/C++ to assembler code instead directly to object code. *TARIFA* instruments the assembler code and the result is translated into object code.

*TARIFA* separates the transaction implementation from the actual transaction usage. Currently we use a self-made transaction library. But using another implementation (like Tim Harris's one) would be hidden by *TARIFA*'s instrumentation.

### 2.1 Assembler Instrumentation

A programmer provides a list with all functions including transactions and functions called from within transactions. We plan to introduce an automated way to generate this list for future versions using static flow analysis. *TARIFA*

```

1      pusha
2      pushf
3      call Tarifa::inTransaction
4      testb %al, %al
5      je .LTARIFA_4_notrans
6      movl 28(%esp), %ecx
7      movl 24(%esp), %ebx
8      leal (%ebx, %ecx, ), %eax
9      pushl $4
10     pushl %eax
11     call Tarifa::read
12     addl $8, %esp
13     popf
14     movl (%eax), %ebx
15     addl %ebx, 28(%esp)
16     popa
17     jmp .LTARIFA_4_end
18 .LTARIFA_4_notrans:
19     popf
20     popa
21     addl (%edx, %ecx), %eax
22 .LTARIFA_4_end:

```

**Figure 2. Instrumenting a memory access in AT&T syntax**

parses the assembler code. It identifies all assembler instructions that contain memory operands. They can be easily detected because they have a special assembler syntax. Today TARIFA instruments all the function’s instructions regardless if they are inside a transaction or not. Whether the original or the instrumented instruction is executed is determined at runtime.

It is important to note that every assembler instruction can be instrumented independently. Figure 2 shows the new code for `addl (%edx, %ecx), %eax` which adds the content of the memory address `(%edx, %ecx)` to the register `eax`. The function names are demangled for comprehensibility. First all registers are saved and it is tested if the instruction is executed within an ongoing transaction. If not the registers are restored and the instruction is executed without modifications. If the instruction is executed as part of a transaction the values of the registers `ecx` and `ebx` are fetched from the stack (because the call of the testing function may modify the registers). The original address of the memory operand is calculated. The address and the size of the memory operand (4 bytes) are passed to `Tarifa::read`. This function returns an address of another memory locations that will act as a cache for the memory operand exclusively for this transaction. After that the `addl` instruction is performed on the cache address.

## 2.2 Evaluation

Even though performance was not a distinct objective we have conducted several tests that not only evaluate the correctness of TARIFA but can be used for performance eval-

uations too. The result is that our small example programs executed as fast as if they were programmed by using other transaction libraries directly. However, on larger examples they will surely outperform TARIFA due to its non-mature implementation.

## 3 Future Work

The next steps address several issues. First of all we want to enhance the performance of the instrumented code. A *static flow analysis* will render the check if the current instruction belongs to an transaction obsolete. Furthermore we plan to switch to a lock-free transaction library [1]. Finally we want to avoid the expensive cache address calculation by employing static analysis and preallocation. We expect that these improvements will make it worth to explore larger examples or even to use TARIFA in productive applications.

Additionally a better integration into C/C++ is required. Currently a programmer must write the code to start and commit a transaction. We want to explore transaction semantics regarding exception handling, failure atomic functions, retry of aborted transactions, and different policies to deal with calls to external functions and system calls.

## 4 Conclusion

TARIFA is a tool for using memory based transactions in C/C++. By hiding the implementation the transaction library is nearly transparent for the programmer. This simplifies the implementation of multi threaded software. We expect that this has a positive influence on applications correctness. Our goal is to make TARIFA deployable into production environments for managing the concurrent accesses to shared data. A more detailed introduction into TARIFA can be found at [4].

## References

- [1] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM Press.
- [2] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.
- [3] Intel. Intel multi-core processor architecture development background. 2005.
- [4] U. Müller. Memory based transactions in C/C++. Study thesis, Dresden University of Technology, 2005. <http://wwwse.inf.tu-dresden.de/AutoPatch/files/theses/mbtccpp-um05.pdf>.