

TRANSACTIONAL MEMORY SUPPORT FOR GCC

1. INTRODUCTION

This document describes the design decision made to implement support for transactional memory in the GCC (version 4.3.2). Our goal is a robust and extensible implementation supporting transactional memory. Featuring several software transactional memory implementations, this implementation should facilitate the use of transactional memory among researchers as well as programmers. Being orthogonal to other parallel programming models is one of our major design goals. For TM implementations based on OpenMP see [6, 2].

The document is structured as follows: First, the roadmap of this project is presented in Chapter 2. To get started the next Chapter 3 describes the usage of the additional features as well as enhancements to the build process. Chapter 4 shows an overview of the data structures. At first the semantics of the new attributes, pragma, and keywords are described. Then, the added GCC internals are documented. Chapter 5 describes the additional compiler passes we plan to implement.

2. ROADMAP

First, the main features:

- (1) Atomic block (rather than begin/end), supporting the closed nesting of transactions
- (2) Weak isolation considering single-lock semantics (see [5] in contrast to strong isolation [7])
- (3) Attributes: `tm_callable`, `tm_pure` (later refinements possible)
- (4) Irrevocable functions (and transactions running in irrevocable mode), with a runtime failure when calling `__tm_abort` after calling such a function in a transaction; also, any access to a volatile variable switches the transaction to irrevocable mode
- (5) Specific syntax to call mangled tm-aware version of a function, useful when developing libraries (e.g., transactional malloc [4])
- (6) Rollback and commit handlers, also useful for libraries;
- (7) Failure atomicity with explicit calls to `__tm_abort`
- (8) C++: classes, inheritance, templates, deferred to a later effort to support C++; but target an orthogonal exception handling when compiling C++ code that interacts with transactional C code. Exception handling will be done according to "Exceptions commit" in [3].

Then, the priorities for the first phase are on (1), (2) without single-lock semantics (meaning that we do not care about quiescence for now), and (3).

In a second phase, we will work on the single-lock semantics (2), (4), (5), (6) and (7). We aim to have a code base that is fully portable across GCC and ICC.

Some time soon, we will look for C++ front-end maintainers to do the front-end extensions.

3. USAGE AND BUILD PROCESS

The compiler switch that has to be set in order to use the transactional memory features is `-fgnu-tm`. Otherwise, the `_Pragma("tm atomic")` is ignored, causing no side effects.

In order to use the transactional properties of the compiler, a supported Software Transactional Memory (STM) library must be installed. The user may specify the location of the library during the configure process of the GCC. The switch added to the configure process is: `--with-stm-lib=PATH` should point to the directory where the STM library is installed. If multiple STM libraries are installed, we suggest to set up multiple GCCs as well. Currently, the TinySTM, and SUN's TL2 are recognised by the configure process. Concerning the compiler implementation we will start with the support for the TinySTM library [1].

4. DATA STRUCTURES

This section covers the attributes and the data structures we plan to implement.

4.1. Attributes. The following list contains the TM-attributes and current GCC attributes having an impact on transactions.

- **const** and **pure** functions do not alter memory locations accessed inside transactions. Still there is an important difference between **const** and **pure**. While **const** functions only evaluate their arguments, **pure** functions may read memory locations. Thus, to prevent reading unsafe values the **pure** functions called from inside transactions are instrumented with calls to the STM runtime library.
- **tm_pure** functions do not read memory locations that are written to from within transactions. Subsequently, functions marked as **const** qualify for **tm_pure**. Further, the programmer can specify functions as **tm_pure** that can safely be called from inside and outside transactions.
- **tm_callable** marks a function as callable from a transaction. The compiler provides a STM-instrumented clone of the function.
- **tm_unknown** **future work** is the default value that relies on the compiler to detect the right context if the called function is compiled by the compiler as well. In case the function call is to a legacy function not known to the compiler, the compiler will generate code that defers the decision if an operation has to be treated as irrevocable until run-time.
- **malloc**: **future work** functions annotated with this attribute are treated as if a returned non-null pointer can not alias any other valid pointer when the function returns. In a transactional context this

implies that it is safe to call it from a function marked as `tm_unknown` and leave the responsibility to the compiler. To call the transactional version of a function a wrap mechanism is used. (see Ali's upcoming OOPSLA paper).

4.2. Keywords.

- `_Pragma("tm atomic")` (or `#pragma tm atomic`) is followed by a compound statement that forms a transaction. Statements within transactions are instrumented by calls to the STM run-time system [8]. This allows for the concurrent execution of transactions while assuring isolation. In order to have portable code between the ICC and the GCC, the `__tm_atomic` keyword is supported, if `-imacros gcc-tm.h` is given as a command line switch. The file `gcc-tm.h` contains a macro expanding `__tm_atomic` to `_Pragma("tm atomic")`. The drawback is that the source code has to be C99 compliant.
- `__tm_abort` allows the user to explicitly abort a transaction. This function call maps directly to the compiler builtin `GTM_txn_abort`. Supporting irrevocable calls alters the handling of aborts: the transactions run in a different mode and can not be rolled back. If an abort is issued while or after executing a call to a irrevocable function before the transaction commits, the runtime system issues a fatal error.

4.3. Trees.

- `TREE_CODES`
 - `GTM_TXN` is set by the parser to mark the beginning of a transaction.
 - `GTM_TXN_BODY` contains the transaction itself.
 - `GTM_RETURN` is set by the parser to mark a possible exit of a transaction. After lowering this nodes are substituted with calls to the STM specific runtime system.
- `TREE_MACROS`
 - `GTM_DIRECTIVE_P` returns true if the tree node is either a `GTM_TXN`-node or a `GTM_RETURN`-node.
 - `GTM_CALLABLE_P` indicates that the corresponding attribute was set for the declaration. Since the function is marked as callable from within a transaction the function will be cloned and a transaction-safe version, as well as a transaction unaware version will be available inside the compiler.
 - `GTM_PURE_P` returns true if the attribute `tm_pure` was specified, indicating that the function has no side effects and may safely be called from within a transaction.
 - `GTM_IS_CLONED_P` returns true if the function has been cloned to allow for transactional properties.

4.4. **Builtins.** This section describes the GCC-builtins. These builtins are mapped to the corresponding STM functions. Depending on the interface

of the STM we may have to add some builtins that define special actions. Currently, only the basic functionality of the STM is represented in builtins.

- `GTM_txn_begin` starts a transaction by calling the STM library function with corresponding parameters.
- `GTM_txn_commit` ends a previously started transaction by calling a STM library function.
- `GTM_txn_abort` is used to abort a transaction by calling the corresponding STM function.
- `GTM_shared_load` instruments a read to a memory location by calling the STM library.
- `GTM_shared_store` instruments a write to a memory location by calling the STM library.

5. PASSES

Besides the necessary modifications in several files of the GCC code base, we aim to implement two passes. First, the `GTM_TXN`, `GTM_TXN_BODY`, and `GTM_RETURN` constructs are introduced while parsing. The construction of the control flow graph is altered according to the OpenMP scheme by splitting basic block, when a `GTM_DIRECTIVE` is encountered, simplifying the access to transactions.

The first pass (`gtm_exp`) does the expansion of transactions and the recombining of the previously split basic blocks. Memory loads and stores are instrumented with calls to the STM runtime. In addition the pass checks for restrictions that apply for transactions. For instance, `__tm_abort` can only be called inside a transaction. In order to have a more convenient access to transactions, a `gtm_region-tree` is built. The region tree facilitates the flattening of transactions. The second pass implements a checkpointing scheme in addition to the `setjmp/longjmp` mechanism. One additional basic block is introduced. This basic block is executed in case of a rollback and restores the values of variables. The saving of the values (and storing them into a temporary variable) is done before calling `setjmp`. In order to reduce the number of copied variables only variables, that are live-in to the transaction are considered. The availability of liveness information requires to write a pass on SSA-form. The `gtm_checkpoint`-pass removes the marker (set by `gtm_exp`-pass) and adds the real checkpointing scheme, illustrated in Figure 1. The instruction sequence before the `setjmp` call captures the value of the live-in variable `a` and saves it into the temporary variable `txn_save_a`. In case the transaction has to roll back, the library executes a call to `longjmp` and returns to the point where the `setjmp` was called with a return value that is not 0. Subsequently, the basic block on the right hand side gets executed and the value is restored to `a`. The Φ node on the next basic block merges the different versions of `a`.

REFERENCES

- [1] <http://tinystm.org/tinystm>. Online, last accessed 30th of May 2008.
- [2] Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun. The `opentm` transactional application programming interface. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and*

- Compilation Techniques*, pages 376–387, Washington, DC, USA, 2007. IEEE Computer Society.
- [3] Lawrence Cowl, Yossi Lev, Victor Luchangco, Mark Moir, and Dan Nussbaum. Integrating transactional memory into c++. In *Workshop on Transactional Computing (TRANSACT)*, 2007.
- [4] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg. Mct-malloc: a scalable transactional memory allocator. In *ISMM '06: Proceedings of the 5th international symposium on Memory management*, pages 74–83, New York, NY, USA, 2006. ACM.
- [5] Vijay Menon, Steven Balensiefer, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, Richard Hudson, Bratin Saha, and Adam Welc. Practical weak-atomicity semantics for java stm. In *SPAA'08: 20th ACM Symposium on Parallelism in Algorithms and Architectures*, June 2008.
- [6] Miloš Milovanović, Roger Ferrer, Vladimir Gajinov, Osman S. Unsal, Adrian Cristal, Eduard Ayguadé, and Mateo Valero. Multithreaded software transactional memory and openmp. In *MEDEA '07: Proceedings of the 2007 workshop on MEMory performance*, pages 81–88, New York, NY, USA, 2007. ACM.
- [7] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in stm. *SIGPLAN Not.*, 42(6):78–88, 2007.
- [8] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code generation and optimization for transactional memory constructs in an unmanaged language. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 34–48, Washington, DC, USA, 2007. IEEE Computer Society.

E-mail address: schindew@ira.uka.de

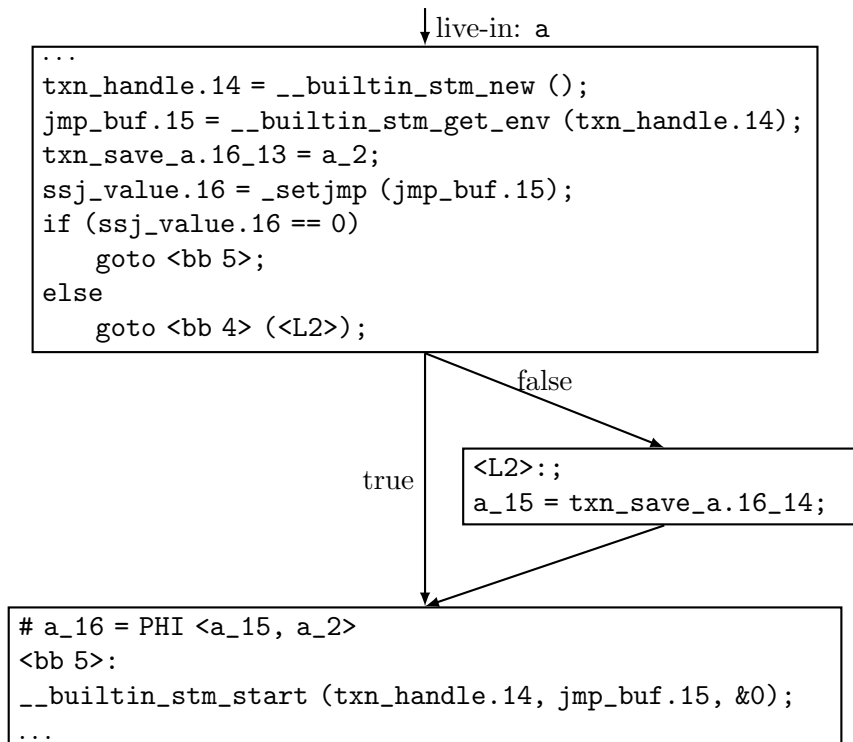


FIGURE 1. Checkpointing mechanism after the `gtm_checkpoint-pass`.