Using Barrier Elision to Improve Transactional Code Generation

BRUNO CHINELATO HONORIO, University of Campinas - UNICAMP, Brazil
JOÃO P. L. DE CARVALHO, University of Alberta, Canada
CATALINA MUNOZ MORALES, University of Campinas - UNICAMP, Brazil
ALEXANDRO BALDASSIN, State University of São Paulo - UNESP, Brazil
GUIDO ARAUJO, University of Campinas - UNICAMP, Brazil

With chip manufacturers such as Intel, IBM, and ARM offering native support for transactional memory in their instruction set architectures, memory transactions are on the verge of being considered a genuine application tool rather than just an interesting research topic. Despite this recent increase in popularity on the **hardware side of transactional memory (HTM)**, **software support for transactional memory (STM)** is still scarce and the only compiler with transactional support currently available, the **GNU Compiler Collection (GCC)**, does not generate code that achieves desirable performance. For **hybrid solutions of TM (HyTM)**, which are frameworks that leverage the best aspects of HTM and STM, the subpar performance of the software side, caused by inefficient compiler generated code, might forbid HyTM to offer optimal results. This article extends previous work focused exclusively on STM implementations by presenting a detailed analysis of transactional code generated by GCC in the context of HybridTM implementations. In particular, it builds on previous research of transactional memory support in the Clang/LLVM compiler framework, which is decoupled from any TM runtime, and presents the following novel contributions: (a) it shows that STM's performance overhead, due to an excessive amount of read and write barriers added by the compiler, also impacts the performance of HyTM systems; and (b) it reveals the importance of the previously proposed annotation mechanism to reduce the performance gap between HTM and STM in phased runtime systems. Furthermore, it shows that, by correctly using the annotations on just a few lines of code, it is possible to

This work was supported by FAPESP, and the Center for Computational Engineering and Sciences (CCES).
Authors' addresses: B. C. Honorio, C. M. Morales, and G. Araujo, UNICAMP - University of Campinas Institute of Computing Av. Albert Einstein, 1251 - Cidade Universitária, Campinas - SP 13083-852; emails: {bruno.honorio, catalina.morales}@ic.unicamp.br, guido@unicamp.br; J. P. L. de Carvalho, Department of Computing Science 2-21 Athabasca Hall University of Alberta Edmonton, Alberta Canada T6G 2E8; email: joao.carvalho@ualberta.ca; A. Baldassin, UNESP - State University of São Paulo Instituto de Geociências e Ciências Exatas - Câmpus de Rio Claro DEMAC - Departamento de Estatística, Matemática Aplicada e Computação Avenida 24 A,1515 - Rio Claro/SP 13506-900; email: alexandro.baldassin@unesp.br.

1544-3566/2022/07-ART46 $15.00
https://doi.org/10.1145/3533318

reduce the total number of instrumented barriers by 95% and to achieve speed-ups of up to 7× when compared to the original code generated by GCC and the Clang compiler.[1]

CCS Concepts: • **Software and its engineering → Compilers**; **Runtime environments**; • **Computer systems organization → Multicore architectures;**

Additional Key Words and Phrases: Transactional memory, debugging

ACM Reference format:
Bruno Chinelato Honorio, João P. L. de Carvalho, Catalina Munoz Morales, Alexandro Baldassin, and Guido Araujo. 2022. Using Barrier Elision to Improve Transactional Code Generation. *ACM Trans. Archit. Code Optim.* 19, 3, Article 46 (July 2022), 23 pages.
https://doi.org/10.1145/3533318

1 INTRODUCTION

The rise of multi-core processors brought to light the precarious state of the tools available to write and debug parallel code [27, 38]. This has led both industry and academia to devote a lot of effort into creating new programming paradigms that better suit the development of concurrent software. One result of such efforts is a mechanism known as *Transactional Memory* **(TM)** [14, 16].

Transactional Memory makes use of the concept of a transaction: a region of code that is executed in an atomic and isolated manner. Therefore, programmers should only worry about defining code excerpts that have shared memory accesses, while the transactional framework handles concurrency control. The **transactional** framework can be realized in **hardware (HTM)**, **software (STM)**, or a mix of **both (HyTM)**, all with different compromises. While HTM allows for better performance, it is generally limited to small transactions where the data workload should not exceed the limits of the hardware's versioning capacity, which is generally limited to the first two cache levels [26, 42]. Although STM does not have a limit regarding the size of transactions, it needs to instrument each and every access to the memory to ensure atomicity and isolation.

Hybrid TM (HyTM) frameworks have been proposed [5, 8, 20, 30] to leverage on the advantages of both HTM and STM, where a dynamic mechanism is used to assign a transaction to STM or HTM according to different characteristics such as size and abort ratio. A similar approach, called **Phased Transactional Memory (PhTM*)** [28], executes transactions in phases, with each phase composed of only one execution mode (e.g., HTM or STM). As a result, there is no coordination between transactions running in different modes like in HyTM, making its implementation simpler and more flexible. Recent studies with a modern PhTM-based system, called PhTM*, have

[1]Extension of Conference Paper: This paper extends our work published at the IEEE International Parallel and Distributed Processing Symposium (IPDPS 2020), titled "Improving Transactional Code Generation via Variable Annotation and Barrier Elision" [11]. Previously, we claimed that transactional code generated by a compiler can significantly hinder the performance of STMs, and we proposed a language annotation mechanism that can be used to effectively remove unnecessary barriers and improve performance. This current paper extends that work by generalizing its claim, showing how over-instrumentation also negatively affects the performance of **Hybrid TMs (HyTM)**, and demonstrating that our proposed language annotation benefits HyTM implementations as well. This extended work reports experiments with a state-of-the-art **phase-based TM runtime system (PhTM*)**.

In particular, we would like to highlight that both Section 2.2, discussing Phased Transactional Memory (PhTM*), and Section 5.3 are completely new content not part of the prior work. Lastly, Section 5.2 not only adds performance numbers for PhTM* but also presents an in-depth discussion of the implications of having TMFree (our language annotation) as a memory barrier elision mechanism in the context of HyTMs.

In short, we believe that our new contribution further advances research on compiler support for transactional memory and provides empirical evidence that supports the claim that the proposed approach for Clang/LLVM is independent of any type of TM runtime system.

shown that phased systems can offer better performance compared to state-of-the-art HyTM implementations [10, 21].

Regarding support, HTM has seen a rise in popularity since 2013 with chip manufacturers such as Intel [42], IBM [26], and ARM [37] offering native support for TM in their instruction set architectures. It is important to note that this hardware support takes a best-effort approach, which means there is no guarantee that a transaction will eventually be committed by the hardware. Therefore, the programmer must provide a fallback execution path to guarantee progress. Despite all the research on transactional memory conducted in the last 15 years, compiling support for it is still rare, with the only mainstream compiler capable of generating transactional code being the **GNU Compiler Collection (GCC)** [1]. This severely limits not only the development of new applications but also the research potential of TM, particularly given that the *de facto* standard for compiler research is the Clang/LLVM compiler infrastructure [24].

The transactional code generated by a compiler makes use of instrumented accesses to shared memory (reads and writes), in the form of barriers, enabling the TM runtime to detect conflicts among transactions and thus to perform concurrency control. As identified previously by Dragojevic et al. [12] and further explored in this article, much of the observed STM inefficiencies come mainly from a phenomenon called *over-instrumentation*. Over-instrumentation occurs when a compiler cannot determine, at compile time, if a variable is shared or not. Since the compiler must be conservative, it will insert barriers even if they are not required. As a result, the runtime performance of STM tends to be subpar. Even hybrid or phased solutions, such as `PhTM*`, will not be able to offer optimal results because of the STM phase. Due to these setbacks caused by over-instrumentation, TM is still not commonly used in the development of commercial concurrent software, which further highlights the relevance (and necessity) of additional research on compiler support for transactions.

In a previous work [11], we assessed the performance improvements delivered when transactional barriers are removed from transaction-private variables, after annotating variables with `TMFree`.[2] However, in that work we considered only the case of STM implementations, and did not evaluate the impact and feasibility of such approach in the context of HyTMs. This paper is an extension of that work which seeks to fill in this gap. Here, we also use the transactional memory support on Clang/LLVM compiler from [11], but we add an evaluation of the proposed approach in the context of HyTM, more specifically `PhTM*` [21]. This article reports experimental results in Section 5 that show speedups for both STM and PhTM runtimes, in contrast to the STM-only approach that we proposed in [11].

1.1 Motivating Example

To illustrate the compiler over-instrumentation issue, consider the code snippet of Figure 1(a). It shows a function that performs the comparison of two strings: `s1` and `s2`. The annotation of line 1 (`TM_SAFE`[3]) declares that this function might be called by transactions (see Section 2.1.1 for details), which makes the compiler generate two versions of the code: the original one with no instrumentation; and an instrumented clone that will be called by transactions during execution. Since the compiler cannot determine at compile time whether the two compared strings can be modified by concurrent transactions, it takes a conservative approach and adds barriers to variables `s1` and `s2`.

Notice, however, that in some situations the strings to be compared are read-only, in which case the instrumentation only adds overhead. If a programmer had a way to tell the compiler that the

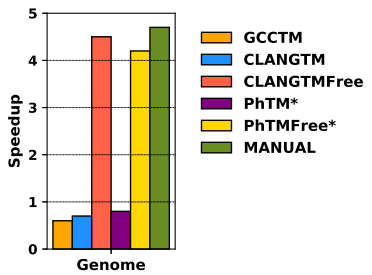
[2]This current `TMFree` keyword was renamed from `TMElide` in [11] as *free* is readily understood, in contrast to the notion of *elision*, which requires programmers to comprehend the underlying compiler implementation.
[3]Macro for the `transaction_safe` function attribute used by GCC's TM support [1].

```
1   TM_SAFE
2   int compare(char* s1,
3               char* s2)
4   {
5       char c1, c2;
6       do {
7         c1 = *s1++;
8         c2 = *s2++;
9         if (c1 == '\0')
10          return c1 - c2
                ;
11      } while (c1 == c2)
            ;
12      return c1 - c2;
13  }
```

(a)

```
1   TM_SAFE
2   int compare(TMFree<char>* s1
        ,
3               TMFree<char>* s2
                    )
4   {
5       char c1, c2;
6       do {
7         c1 = *s1++;
8         c2 = *s2++;
9         if (c1 == '\0')
10          return c1 - c2;
11      } while (c1 == c2);
12      return c1 - c2;
13  }
```

(b)



(c)

Fig. 1. (a) Non-annotated version of the compare function; (b) Its corresponding annotated version using the proposed language support (highlighted in red); (c) Speedup comparison using the code generated by GCC (GCCTM) and Clang/LLVM with (CLANGTMFree) and without (CLANGTM) the proposed annotation, phased runtimes with (PhTMFree*) and without (PhTM*) the annotation, and finally, manually instrumented code (MANUAL).

strings are read-only, the compiler would not add barriers to them. As a matter of fact, Genome is an application that makes an intense usage of this function (≈2 billion calls during sequential execution), and thus considerably suffers from over-instrumentation side-effects. Adding the proposed TMFree annotation, as shown in lines 2–3 of Figure 1(b), forces the compiler to elide the barriers associated to variables s1 and s2 (lines 7–8). Figure 1(c) shows the resulting speedup with respect to Genome sequential execution, when barriers are elided for 18 threads using the following code generation strategies: (i) GCCTM, automatically instrumented by the GCC compiler; (ii) automatically instrumented by Clang/LLVM with (CLANGTMFree) and without (CLANGTM) the proposed language support and STM runtime; (iii) similarly to the previous case, but using the PhTM* runtime (PhTMFree* and PhTM*, accordingly); and (iv) MANUAL, manually instrumented code.

As Figure 1(c) makes evident, leaving the instrumentation exclusively to the compilers might incur in high overheads, since their conservative approach to instrumentation leads to a slowdown of 0.5× when compared to the sequential execution time. Meanwhile, correctly annotating the two variables s1 and s2 using the TMFree language support with either the pure STM or PhTM* runtimes, produces speedups of up to 4.4× for the Genome application, nearly matching the performance of the manually instrumented code. Also notice that the PhTM* results show that, without the TMFree support, the implementation alone is not able to achieve significant improvements over the pure STM runtime results. As shown in Section 5.3, Genome stays predominantly in software mode for PhTM*, but using the TMFree annotation (PhTMFree*) the application improves its performance, as it can take advantage of the phased approach to stay more time in the (faster) HTM mode.

1.2 Contributions

This article extends our previous work in [11] and together make the following contributions[4]:

- It proposes the first transactional memory support for the Clang/LLVM compiler framework that works as a drop-in replacement of GCC (see Section 3). The extensions added to Clang/LLVM generate code following Intel/GCC's common TM API;

[4]The last two contributions are only part of this extended version.

- It proposes a novel language annotation mechanism (`TMFree`) that allows programmers to instruct the compiler to elide barriers for certain memory regions (see Section 3.2);
- It presents a profiling tool that ranks the memory regions that were accessed the most at run-time, thus unveiling barrier elision opportunities not easily detected by programmers (see Section 4);
- It shows the performance gained by using barrier elision through variable annotation for some STAMP benchmarks for both pure STM and Phased TM runtimes, achieving speedups of up to 7× (see Section 5);
- It reveals that, although some applications can benefit from HTM in phased runtimes, other applications do require STM due to their large storage capacity requirements, and therefore can take advantage of the `TMFree` annotation to reduce the performance gap w.r.t. GCC's transactional code (Section 5.3).

The rest of this paper is organized as follows. Section 2 lists background material and previous related work. Section 3 details the new transactional memory support added to the Clang/LLVM compiler framework and the proposed annotation mechanism that implements transactional barrier elision. Section 4 describes the proposed profiling tool and how it was used to identify barrier elision opportunities. Section 5 details the experimental setup and analyzes the performance improvements achieved by the proposed approach, when compared to manual code and the GCC's strategy, as well as discusses the impact of such performance improvement on a Phased TM implementation. Finally, Section 6 concludes the work.

2 BACKGROUND

Transactional Memory (TM) [14, 16] provides transparent concurrency control through the abstraction of *transactional blocks*. A transactional block groups instructions that must take effect atomically; this means that all speculative changes must be committed at once, in which case all variable updates within the transaction are committed to memory, or discarded (i.e., *aborted*) entirely. Back in 1993, Herlihy and Moss introduced the concept of transactional memory by devising a hardware support which added new instructions and also extended the cache coherence protocol [15]. Since then, the area has flourished alongside the wide adoption of multi-core processors. Despite initially being introduced as a hardware mechanism (HTM) [15], most of the following research focuses on software systems (STM), more specifically on library-based implementations [9, 13, 29]. STMs, and library-based approaches are, in general, more prominently used due to their flexibility. In addition, commodity processors with HTM support only reached the market in 2012. The release of microprocessors providing hardware transactions by companies such as IBM [26] and Intel [42] renewed interest in the area. Recently, HTM support for ARM processors was announced [37], showing the possibility of bringing TM closer to the mobile and embedded world.

Other implementation strategies, such those presented by Dalessandro et al. [8], Lev et al. [28], and Carvalho et al. [10], have focused on taking advantage of both hardware and software. In hybrid systems (HyTM), transactions can be executed concurrently using both hardware and software support, whereas phased systems (PhTM) split the execution in phases. In a phase, all transactions execute in the same mode (e.g., hardware or software), thus, simplifying the implementation when compared to HyTM as noticed in [21]. Even with all the advances and improvements seen in the last decade on TM research, works on programmability aspects are limited [2, 6]. This article aims to start filling this gap by providing new opportunities for performance enhancement through barrier elision in the Clang/LLVM compiler. Library support, although suitable for research, requires programmers to manually use the transactional API to instrument accesses to shared memory, lowering the level of abstraction that TM aims to achieve. This requirement creates a steep learning

curve for newcomers because extremely careful attention must be paid to when transactions are manually instrumented in order to avoid incorrect or unnecessary instrumentation.

Support for TM in unmanaged languages only recently started to appear in the GNU Compiler Collection (GCC) [1]. Although Intel offered compiler support from 2008 to 2012, it was never part of its main compiler (ICC) and is now discontinued. Support for TM in the C language dates back to 2014 in GCC (version 4.7) and, for C++, to 2016 (version 6.1). GCC's runtime support is heavily based on Intel®'s TM ABI [19], which is also used in the added support proposed in this work for Clang/LLVM (see Section 3).

2.1 Transactional Memory Support in GCC

The transactional support provided by GCC [1] is composed of three parts: (i) language support to annotate transactional blocks and functions (Section 2.1.1); (ii) transactional code generation that inserts memory access barriers and other calls to the transactional runtime (Section 2.1.2), and (iii) the runtime library itself shipped with the compiler (Section 2.1.3). Compilers that support TM for other languages than C/C++ exist, but GCC, to the best of our knowledge, is the only C/C++ compiler that is still maintained which support TM. Intel offered a prototype C/C++ compiler that supported STM but it was discontinued in 2010. As `TMFree` is a C/C++ extension, GCC is the only available baseline used for comparison in Section 5.

2.1.1 Language Support. Transactional blocks only allow calls to `TM_SAFE` and `TM_PURE`[5] functions. Functions marked with `TM_SAFE` are cloned by the compiler, producing a copy of the original function with all memory accesses instrumented with runtime calls. As for `TM_PURE` functions, the compiler assumes that they do not have side-effects, thus, no instrumentation is performed. If a function has neither the safe nor the pure attribute, it is considered *unsafe*. GCC also supports unsafe function calls by means of *relaxed* transactions.

2.1.2 Transactional Code Generation. For each transactional block, GCC generates two copies of the code within the block: one with the memory accesses instrumented with the runtime library, and another without any instrumentation that can be used, for instance, with HTM, or other alternative synchronization mechanisms.

During code generation, the compiler might not be able to determine which barriers to elide due to problems such as pointer aliasing [22]. This impossibility forces the compiler to conservatively add unnecessary barriers, leading to the phenomenon known as *over-instrumentation* [12].

In order to see that the over-instrumentation problem can happen even with very simple code, consider the example shown in Listing 1 from Honorio et al. [17]. In this example, `foo` is a function that starts a transaction (line 2) which first creates a linked-list (line 3) and then calls `initList`, passing the list as an argument (line 4). After some processing with `L`, the transaction publishes some elements of `L`. Notice that the elements of `L` will only become visible to other transactions when, and if, the publisher transaction commits. Therefore, all elements of `L` could be accessed in `initList` without barriers. However, as `initList` is annotated with the attribute `TM_SAFE`, the compiler will create an instrumented version of the function. The loop from line 13 through 16 walks through the linked-list, zeroing out each element (line 14). From the point of view of `foo`, the linked-list `L` is only local but, since the compiler cannot deduce that, all read and write accesses to the list in `initList` will be done via barriers.

Listing 2 shows the simplified assembly code generated by GCC for the example in Listing 1. Lines from 1 to 11 refer to the procedure `foo`, while lines 12 to 32 refer to the procedure `initList`.

[5]Used via `__transaction_safe` and `__transaction_pure` function attributes introduced by the original TM ABI from Intel [19].

```
1   void foo() {
2     __transaction_atomic {
3       list_t *L = createList();
4       initList(L);
5       // some processing with L
6       // publish elements of L
7     }
8   }
9
10  TM_SAFE
11  void initList(list_t *l) {
12    node_t * ptr = l->head;
13    while (ptr != NULL) {
14      ptr->v = 0;
15      ptr = ptr->next;
16    }
17
18  }
```

Listing 1. Example of over-instrumentation.

In this code fragment, the highlighted words indicate the calls to the transactional ABI to start a transaction (line 2), allocate the linked-list (line 4) and commit the transaction (line 9). The read barriers to access the head of `L` (line 14) and the `next` field (line 26), as well as the write barrier to reset the field `v` (line 23) are also in boldface. This example shows that, in fact, the compiler instruments all accesses to the linked-list `L`. The compiler is unable to infer that the list is private to the transaction and therefore inserts transactional barriers to every load and store inside `initList`. This problem can be avoided by simply adding the `TMFree` type qualifier to the argument of `initList`, as discussed in Section 3.

2.1.3 Transactional Runtime Library. GCC's transactional runtime closely follows Intel's TM ABI [19]. It generates transactional code to start and commit transactions using the ABI functions `_ITM_beginTransaction`, `_ITM_commitTransaction`, respectively. The runtime is implemented as a dynamic shared-library, thus enabling the user to switch between different runtime implementations without recompiling the application's code. In addition, the ABI also defines the load and store barriers for each basic data-type in C and C++ that is supported by the target architecture. For instance, function `_ITM_RU8` reads a 64-bit value. In this work, a runtime following the same TM ABI was implemented and is used by the code generated by both GCC and Clang/LLVM. By using the same library, performance differences due to implementation details alone are avoided.

2.2 Phased Transactional Memory* (PhTM*)

A phased TM implementation (PhTM) has two or more transaction execution modes. The execution mode, as well as the transition between them, are determined by a controller or automaton. The optimal execution mode is selected based on certain characteristics such as transaction size and abort reason. In particular, the phased implementation has the main feature that, unlike other hybrid implementations, it executes all concurrent transactions in a single-mode manner. PhTM implementations were first explored by Lev et al. [28] and later improved by Carvalho et al. [10, 21], who proposed PhTM*.

PhTM* has three modes of execution, namely **hardware (HW)**, **software (SW)**, and sequential (GLOCK). In HW mode, all concurrent transactions are executed in HTM. If there is a persistent

```
1   foo:
2     call  _ITM_beginTransaction
3   # ...
4     call  createList
5   # ...
6     call  initList
7   # some processing with L
8   # publish elements of L
9     call  _ITM_commitTransaction
10  # ...
11  # ...
12  initList:
13  # node_t* ptr = l->head
14    call  _ITM_RU8
15  # if ptr == NULL, goto RETURN
16    testq %rax, %rax
17    je  .L9
18    movq  %rax, %rbx
19  WHILE:
20    leaq  %rbx, %rdi
21  # ptr->v = 0;
22    xorl  %esi, %esi
23    call  _ITM_WU4
24  # ptr = ptr->next;
25    movq  %rbx, %rdi
26    call  _ITM_RU8
27  # if ptr != NULL, goto WHILE
28    testq %rax, %rax
29    movq  %rax, %rbx
30    jne WHILE
31  RETURN:
32    retq
```

Listing 2. Assembly code generated for the example of Listing 1.

hardware capacity abort during execution, the automaton forces a transition to SW mode where there is no capacity restriction. On the other hand, if there are repeated transaction aborts due to conflicts, the automaton triggers a transition to a **global lock mode (GLOCK mode)**, where the atomic region will be executed sequentially. In the SW mode, an STM implementation like NOrec [9] is in charge of the execution progress. The automaton decides to change from SW to HW mode once the conditions related to capacity aborts are overcome. Finally, in GLOCK mode, a global lock is acquired by one thread to sequentially execute a transaction that has been aborted due to conflict with other transactions. The PhTM* automaton can make four types of transitions as shown in figure 2: Ⓐ The $HW \rightarrow SW$ transition occurs when a certain number of capacity-related aborts is reached. Ⓑ The $SW \rightarrow HW$ transition occurs if the aborted transactions that caused the transition to SW have been committed, and the size of the transactions that are being executed are below a threshold. Ⓒ Transition $HW \rightarrow GLOCK$ happens when a transaction persistently aborts due to conflict with other transactions after a number of execution attempts. Finally, Ⓓ transition $GLOCK \rightarrow HW$ occurs once the conflicting transaction has been sequentially executed and committed.

One major benefit of using the proposed annotation with phased TM systems is that it will potentially reduce the amount of time spent by the transactions in SW mode, thus aiding the

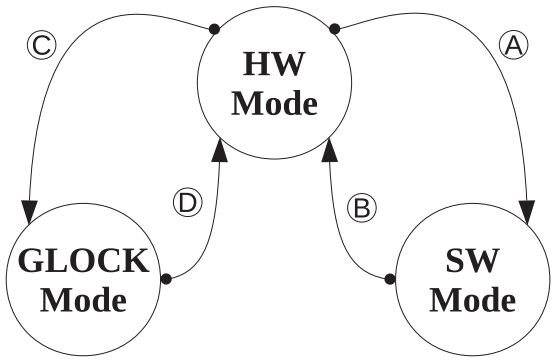


Fig. 2. PhTM* transition automaton.

PhTM* heuristics to switch more frequently to the fastest HW mode, and improving the overall performance.

2.3 Related Work

The challenges of optimizing transactional code in unmanaged languages (C/C++) were first discussed by Wang et al. [40]. They focus on inlining STM's hotspot routines, such as checkpointing code, and describe optimizations to eliminate redundant barriers. Bronson et al. [4] identified opportunities to elide barriers in non-transactional code. These barriers are placed in order to ensure strong isolation between transactional and non-transactional code. Guided by runtime information, Bronson et al.'s approach is able to dynamically optimize isolation barriers. Our work targets memory access barriers on transactional code and is thus orthogonal to Bronson et al.'s proposal. Moreover, `TMFree` enables barrier elision at compile time instead of run time.

The over-instrumentation problem was first characterized by Yoo et al. in [43]. A new construct was proposed to enable programmers to instruct the compiler to elide all barriers within the specified block of code (e.g., a function). Our work proposes an elision mechanism that is more flexible than Yoo et al.'s by enabling elision in the granularity of individual variables. In addition, Yoo et al. do not provide a profiling mechanism to identify opportunities for barrier elision, as in this work. The work by Lev et al. [27] was among the first to recognize the scarceness of debugging tools for TM. Wu et al. [41] discuss optimizations for both managed and unmanaged environments using the IBM XL compiler and the Java virtual machine. The exploitation of platform specific characteristics for better code generation was investigated by Ruan et al. [35]. Their work assessed the interplay between compiler instrumentation and performance regarding which optimizations to perform.

Dragojević et al. [12] extended the previous work of Yoo et al. [43] by also eliding barriers in captured memory accesses, i.e., memory allocated inside transactions which are only globally visible after the allocated transaction's commit. A *capture analysis* is developed for both the runtime and the compiler, similar to the one implemented in the Clang/LLVM support presented in this work. Carvalho and Cachopo later extended the capture analysis for a managed environment based on a Java virtual machine [7].

The most recent work on memory transactions language support is from to Zardoshti et al. [44]. Different from this work, Zardoshti et al.'s do not implement transactional blocks as first-class citizens of C/C++ languages. Instead, they rely on anonymous function attributes to specify the scope of transactions. All the machinery for creating transactional clone functions and inserting memory access barriers is performed via an LLVM plug-in. An extension to enable transactional

constructions in C++ was also proposed by Zardoshti et al. [44]. The proposed extension relies on C++ lambda functions and thus does not work with C programs. Moreover, Zardoshti et al.'s implementation inserts calls to a custom TM runtime API which – unlike the Clang/LLVM's extensions proposed in this work – is not compatible with GCC/Intel's common TM API. In addition, existing TM runtimes that adhere to the common GCC/Intel API would need to be changed to follow Zardoshti et al.'s non-standard API. On the other hand, our work makes transactional blocks part of the C/C++ language in the Clang compiler. In addition, it provides a barrier elision mechanism to avoid over-instrumentation through an LLVM pass. Moreover, Zardoshti et al.'s transactional runtime does not follow Intel's TM ABI [19], thus requiring that existing transactional libraries be adapted prior to their use. Honorio et al. [17] propose a pragma-based elision mechanism that requires programmers to insert pragmas at each usage scope of a transactional local value. Our approach using `TMFree` enforces the correct interplay between local and non-local transactional variables, thus allowing the same flexibility of `elidebar` [17], but with the additional type enforcement guarantees.

Titos-Gil et al.'s work propose hardware and software extensions to relax the kinds of conflicts that are detected in HTM [39]. The annotations proposed by Titos-Gil et al. is orthogonal to `TMFree` as it is designed to optimize barriers that cannot be elided and `TMFree` is used to elide unnecessary barriers. Both `TMFree` and Titos-Gil' extensions can be used together, however, because Titos-Gil' results were collected in a simulation environment, combining the two extensions was not considered for the presented work. The results in Section 5 reflect the performance on real hardware and the use of production ready compilers. Unlike `TMFree`, Titos-Gil et al. do not offer a tool to help programmers identifying optimization opportunities and programmers are left unguided on the more complex task of identifying data dependencies.

The solution proposed in this work relies on programmers to inform the compiler, by means of the `TMFree` annotation, which shared memory locations should not be instrumented. Although at first sight this could be viewed as an extra burden for programmers, the results obtained with the guidance of a novel profiling tool shows that just a couple of annotations enables significant overhead elimination (see Section 5).

3 TRANSACTIONAL MEMORY SUPPORT IN CLANG/LLVM

This section presents the transactional memory support added to the Clang/LLVM[6] compiler framework [24], the *de facto* standard in compiler research and optimization. First, the foundation for transactional memory code generation is discussed (Section 3.1), then the variable annotation mechanism that enables barrier elision is presented (Section 3.2).

3.1 Transactional Code Generation

The foundation for transactional code generation added to Clang/LLVM follows Intel®'s ABI [19] and is entirely compatible with the transactional runtime system shipped with GCC. The parser was extended in Clang to produce a new statement node in the AST[7] (`TransactionAtomicStmt`) whenever a transactional block is found. The other extension aims to recognize the `TM_SAFE` and `TM_PURE` function attributes and add them to the generated **intermediate representation (IR)**. Contrary to GCC, in Clang the instrumented and uninstrumented code paths are created in the AST. In GCC this is done after GENERIC, GCC's AST-like tree-based representation, is lowered to GIMPLE, an IR-like three-address representation. The decision of performing this dual-path

[6]Full source code available at https://github.com/jaopaulolc/TMFree.
[7]Abstract Syntax Tree.

code creation in Clang's AST was made to allow static analysis and provide early feedback to the programmer through error/warning messages (e.g., unsafe calls inside `TM_SAFE` functions).

After the LLVM IR is generated, and if the flag *-fgnu-tm*[8] is provided, the LLVM pass manager schedules the passes in the implemented transformation library. The library implements the following passes:

- `DualPathInfoCollector` – collects transactional boundary and dominance information from all transactional basic-blocks in a function. This is the analysis pass used in all the following transformation passes;
- `TransactionSafeCreation` – a pass that creates a transaction-safe clone of functions called inside transactional blocks or other transaction-safe functions. This pass also creates a global map that associates transaction-safe to their non-safe counterparts. The map is used to handle indirect function calls;
- `ReplaceCallInsideTransaction` – as the name indicates, this pass replaces all calls inside transactions or transaction-safe functions with calls to their transaction-safe counterparts. This includes some non-user functions, such as `memcpy`, `memmove` and `malloc`/`free`[9];
- `LoadStoreBarrierInsertion` – replaces load/store instructions with calls to TM runtime barriers. This pass can elide unnecessary barriers for variables annotated with the `TMFree` qualifier (see Section 3.2);
- `Cleanup` – used to remove any unnecessary code inserted only with the purpose of facilitating code analysis and transformations.

The boundary and dominance analyses allow the barrier insertion pass to elide barriers on captured memory accesses whenever it is possible to prove it at compile-time. The elision of captured memory accesses is responsible for the performance obtained in one of the evaluated applications (see Section 5). In addition, by following the same mechanism used by GCC to store the transactional clones' map, the code generated with the modified Clang/LLVM can be used out-of-the-box with the dynamic-linker from GNU's libraries. In the next section, the barrier elision mechanism syntax and implementation are detailed.

3.2 Barrier Elision Mechanism

LLVM's intermediate representation language is a three-address **SSA (Static Single Assignment)** language that was designed with the purpose of facilitating the writing and reusability of compiler analyses in mind. Nevertheless, since the LLVM IR must be generic enough to allow the representation of programs translated from a plethora of languages, information from certain languages or specific application domains might be lost in the process. Most language communities, such as Rust and Julia, realized the semantic gap between the AST and LLVM IR [25] and dealt with it by employing an extended version of the LLVM IR. This work narrows the semantic gap between Clang's AST and LLVM IR by using LLVM IR metadata to pass language-level information to the transformation and analysis passes. Although the metadata is sufficient to implement the proposed elision mechanism, it is not hard to see that a TM-specific IR could allow elimination of other bottlenecks, such as the cost associated with the lookup of safe cloned function addresses at each indirect function call [43]. In this work, we changed the runtime to save the last function indirectly called, both for GCC and Clang in order to avoid execution time variance due to transactional clone lookups.

The proposed mechanism that enables programmers to elide unnecessary barriers is based on variable annotation, more specifically through a new type qualifier called `TMFree` for the C/C++

[8]The same flag used by GCC was adopted for compatibility reasons.
[9]These functions are implemented as part of the transactional runtime.

```
1  int foo(int* addr) {
2    TMFree<int>* counter = addr;
3    __transaction_atomic {
4      // ...
5      (*counter)++;
6      // ...
7    }
8  }
```

Listing 3. Annotating a TMFree variable in C++.

language. TMFree follows the design of the C11/C++11 Atomic qualifier, which specifies variables that must be accessed atomically. Variables qualified with TMFree are considered local to the surrounding transaction and do not require load/stores barriers to be accessed. Listing 3 shows how a variable can be qualified with TMFree. In the example of Listing 3, let us assume that the memory pointed by addr is private to the thread. Such information is only known by the programmer; the compiler would have to assume that counter (addr) can point to any variable in the program. Therefore, the compiler would not be able to elide the barriers for the accesses to counter. However, by annotating it with TMFree, the information is propagated through the type system. Notice, however, that without the metadata added to the load in the IR instruction, the information about counter being transaction local would be lost.

The proposed annotation mechanism has the benefit of allowing individual variables to have their barriers elided. It does not impose that barriers for all variables in a code block are elided, as in Yoo et al.'s [43] proposed construct. In addition, it enforces that all uses of the transactional local variables are elided by employing a strongly-typed system approach. For instance, in Listing 3, even if the value of counter is passed as an argument to a safe function inside the transactional block, the compiler would require the argument to be TMFree qualified. This enforcement done by the compiler is more robust than Honorio et al.'s [17] approach, which would require the programmer to insert pragmas at each usage scope of a transaction local value. If a previously declared TMFree variable must be accessed transactionally in a given transactional block, but not in another, the programmer can simply add a cast operation to the variable, removing the qualifier. This allows the same flexibility of elidebar [17], but with the additional type enforcement guarantees.

Programmers can make mistakes when using TMFree to elide transactional barriers. TMFree was designed and must be used under the same terms-of-use contract of *restrict*, a standard C/C++ attribute which indicates that a pointer does not alias with others in its declaration scope. Modern compilers do not check if *restrict* was incorrectly used by programmers, as such checks require prohibitively expensive static or run-time analysis [33]. In fact, both flow-sensitive and insensitive static alias analysis are proven NP-Hard problems [18, 23]. Therefore, in the proof-of-concept extensions presented in this work no checking mechanism was added to LLVM to detect incorrect usage of TMFree.

TMFree, while similar in some aspects to the aforementioned *restrict* and other standard C/C++ attributes such as *const*, aims to solve the problem of over-instrumentation in a general use case, whereas the existing attributes are not fit to solve the over-instrumentation problem, or would do so in very specific cases. For example, if a variable is annotated with *const*, any writes to it are a compiler-time error. Thus, it could only be used instead of TMFree if all uses of that variable are read operations. However, for any two non-simultaneous transactions, if the variable is read-only in one transaction but read/write in another, then annotating it with cost would be a compile-time error. This scenario exists in the Genome application. Therefore the *const* qualifier cannot be used in all cases where TMFree would be applicable.

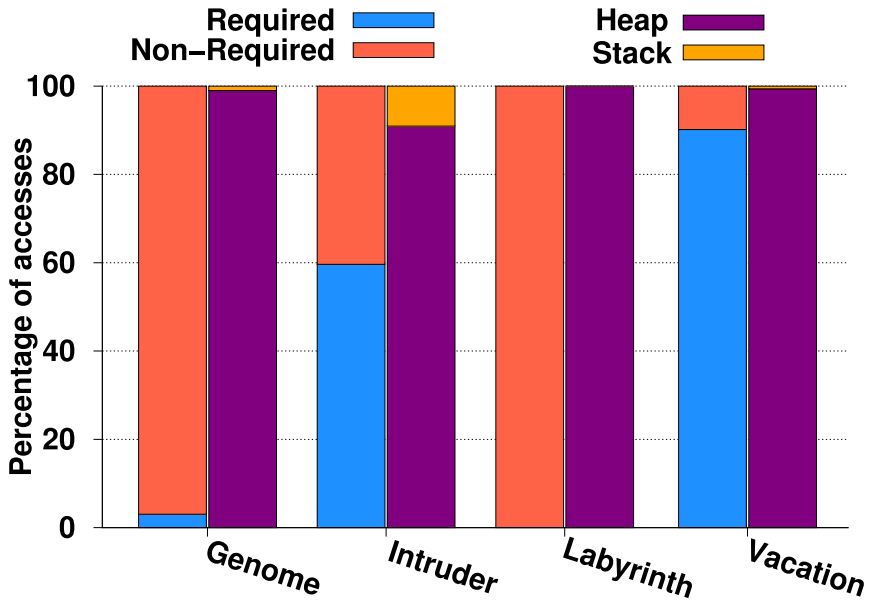


Fig. 3. Classification of memory accesses for STAMP (reads and writes).

On the other hand, if a pointer is annotated with *restrict*, the compiler is safe to assume that the addresses accessed through the pointer do not overlap with any other variable. Thus, if two pointers are annotated with *restrict*, the compiler can assume that access through both pointers will not conflict with each other. However, any two simultaneous transactions might still access the same memory address via the same pointer, causing the compiler to still need to conservatively add the barrier regardless of the *restrict* attribute presence. If the pointer is only used by simultaneous transactions to access non-overlapping memory, then the barriers can be elided with `TMFree`. Therefore the *restrict* attribute is also not a viable replacement for `TMFree`.

4 IDENTIFYING BARRIER ELISION OPPORTUNITIES

Transactional Memory aims to provide transparent concurrency control by removing the burden of synchronization from the programmer. An ideal TM implementation should therefore require only the specification of *what* needs to be synchronized and not *how* to implement it. It is then clear that compiler support for TM is highly desirable, because otherwise the programmer would have to explicitly identify every variable that needs to be consistently accessed. Nevertheless, as the results in the previous section showed, transactional code generated by compilers can be significantly slower than manually instrumented transactions. In the absence of information, compilers need to make conservative decisions that produce over-instrumented code. The required information that would enable compilers to not over-instrument code, e.g., aliasing information or how and where variables were allocated, is easily accessible and known by the application's programmer. In this direction, this work proposes a profiling tool that aims to couple runtime collected information with the programmer's knowledge of the code to enable simple but effective barrier elision.

Previous works [12, 17] focused only on quantitative barrier analysis in transactional code. Figure 3 shows a similar classification of the memory accesses that can be found in the literature for the evaluated STAMP applications. For each application two stacked bars are shown. The stack on the left divides the total number of memory accesses into two types: those that require

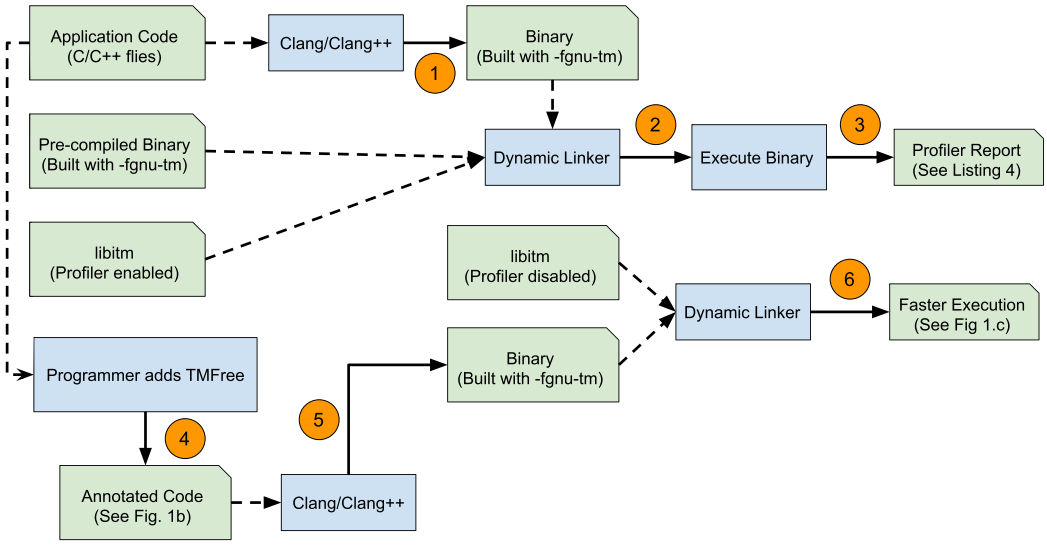


Fig. 4. Illustration of how the profiler report is obtained and TMFree is used to elide unnecessary barriers.

```
============== STACK READS ===============
PC: 405755 Count:  41919668
  Source file: genome/sequencer.c
  Function: _ZGTt14compareSegment
  Source line: 184

============== STACK WRITES ==============
PC: 406b0c Count:  16777216
  Source file: genome/../lib/hashtable.c
  Function: _ZGTt18TMhashtable_insert
  Source line: 754

=========== HEAP READS ============
PC: 40578e Count:  2190204004
  Source file: genome/sequencer.c
  Function: compare
  Source line: 167
```

Listing 4. Output of proposed profiling tool for Genome.

a transactional barrier (Required) and those that were unnecessarily accessed through barriers (Non-Required). All manually annotated accesses were assumed as required, following the same methodology of previous works [12, 17]. In addition, the profiling was performed only on the single-threaded execution for each application. As first noted by Dragojević et al. [12], this assumption might overestimate the number of barriers, however, for the purpose of the discussion in this section, they are reasonable enough. The second stack of bars (right) divides the accesses into Heap and Stack, respectively.

Figure 4 illustrates how simple it is to use the proposed profiling mechanism, integrated into the TM library runtime (libitm), to elide unnecessary barriers using TMFree. Green boxes represent input files or output generated by a tool/process. Blue boxes represent the execution of a tool/process. Dashed arrows indicate the consumption of a file or an output of a tool/process. Solid arrows represent the execution of a tool/process. First, a C/C++ application is compiled (1) using the

modified Clang/Clang++ tool to produce a binary.[10] Then, the generated binary is linked ② against GNU's TM runtime augmented with the profiling mechanism proposed in this work. Finally, the profiler's report is generated once the binary is executed ③. Guided by the straightforward report (See Listing 4), the programmer annotates the identified variables with `TMFree` (See Figure 1(b)) ④. Then, the annotated code is compiled ⑤ with the modified Clang/Clang++ tool to produce a binary without transactional barriers for the annotated variables. The produced binary, once linked ⑤ against GNU's TM runtime, now with the profiling mechanism disabled, will contain direct load-/store operations for access to annotated variables. If the programmer identified key unnecessary barriers as in Figure 1(b), then the produced binary should execute faster as Figure 1(c) indicates.

Figure 3 data was collected with the proposed tool and a debug-enabled version of GNU's TM library (libitm). This debug version has the same API as the non-debug runtime. But instead of implementing the TM algorithm for concurrency control and conflict resolution, each function that access application data and allocates memory was instrumented. The instrumentation code collects the range of the transactions stack memory addresses by calling the compiler built in function `__builtin_dwarf_cfa`,[11] before starting and at the moment each read/write barrier is called. This simple mechanism allows the identification of a memory access to the transaction's local stack. Captured memory analysis [12] is implemented by recording the dynamically allocated memory performed inside of the transaction at runtime. Memory accesses that reference captured memory are thus classified as heap accesses.

Figure 3 clearly shows the over-instrumentation problem, however, it does not give any information to programmers on where exactly the unnecessary barriers are in the code. In order to guide programmers on which barriers to elide and where in the code to insert the `TMFree` annotation, we also developed a profiling tool. The tool is designed to be used as a Profile-Guided Optimization tool to collect the application's runtime information. The tool consists of just a single shared library that implements the same API of GNU's TM library, extending its basic functionality to track barrier execution frequency at run time. For each barrier, the tool records its precise source location: file name, function that it belongs to and line number in the source code. The precise source location is already encoded in the binary by most production ready compilers (e.g., GCC and Clang) when the flag `-g` is used, thus it does not required additional metadata nor any changes to the debugging capabilities of the compilers. The location information is stored in the debug information section of the binary, together with the `TMFree` metadata (see Section 3.2). By design, the tool can be transparently used with precompiled transactional code. It can also be employed incrementally, meaning that the programmer can execute the application code using the tool, elide some barriers using the proposed annotation mechanism (see Section 3.2) and the tool's report, to then re-execute the application with the elided barriers for further barrier elision.

Listing 4 shows part of the tool's output for `Genome`. Besides classifying each barrier according to its access type (read or write) and location (stack or heap), the tool outputs the barriers in execution frequency order, from the most to the least executed. The tool obtains the debug information using the **BFD (Binary File Descriptor)** library from the GNU project, as done by other tools such as **GDB (GNU Debugger)**. Therefore, the profiling tool works out-of-the-box with applications compiled for Linux and does not require any changes to the binary format or any special support. The BFD library works with a plethora of binary formats and the debug information used by the tool is format agnostic. All experimental results with ClangTM+`TMFree` and `PhTM*` +`TMFree` presented in Section 5 were obtained after collecting reports like those shown in Listing 4 and annotating the respective variables with `TMFree`.

[10]Linking against a precompiled binary is supported and will also generate the profiler report.
[11]Returns the *Canonical Frame Address* of the callee (a.k.a. call frame address).

Dragojević et al. [12] proposed a mechanism to elide barriers to both stack and heap captured memory at run time. Nevertheless, based on the overhead added to the application[12] when the debug-enabled version of the transaction was used for profiling, it is hard to see how the overall performance would not suffer, as claimed in their work [12]. In addition, the results in the following section show that by just annotating a few lines of code, following a very straightforward methodology using the proposed tool, it is possible to achieve significant performance improvement without adding any run time overhead. The feedback provided by the proposed tool coupled with the programmer's knowledge of the application can enable barrier elision, even in situations where the compiler would not be able to do so alone, due to insufficient, or unavailable, information (e.g., aliasing). In this sense, the `TMFree` mechanism guided by the above described profiling tool can be more effective than most runtime-based or purely compiler-based solutions.

5 EXPERIMENTAL RESULTS

This section shows the experimental results for STAMP using our variable annotation mechanism (Section 5.2). We also show the performance improvements when coupling `TMFree` to a pure STM implementation, as well as the `PhTM*` implementation, where both HTM and STM are leveraged. First, we start with a description of the experimental setup.

5.1 Experimental Setup

The experimental results presented next contrast our novel TM support added to Clang with GCC's support [1]. As discussed previously (Section 3.1), both compilers generate transactional code following Intel®'s TM ABI [19]. In order to avoid performance differences due to different library implementations, the same runtime library was used for the code generated by both compilers. The runtime library employs `NOrec` as the STM system [9]. `NOrec`'s code was developed by the Rochester Synchronization Group and released as part of the RSTM package [29]. `NOrec` employs a global sequence lock, performs deferred versioning and eager conflict detection by means of value-based validation. We refer to the performance of the code generated with Clang and GCC as `CLANGTM` and `GCCTM`, respectively. Moreover, performance issues induced by the memory allocator [3] were avoided by using the TCMalloc allocator with the changes suggested by Nakaike et al. [32].

The results reported in this section represent the mean of 20 executions.[13] All applications and TM implementations were compiled with GCC (GNU C/C++ Compiler) 7.3 and Clang 6.1, whenever our novel Clang TM support was used. We observed a performance degradation on transactional code generated by GCC when vectorization was enabled. Therefore, all applications were compiled with optimization level three (-O3) and vectorization disabled.[14] In this section, we present experiments conducted on a machine powered by a Intel® Xeon™ 5220 2.2GHz processor and 188GB of RAM. The processor has 18 physical cores and 2 hardware threads per core (total of 36 SMT threads). The machine runs CentOS 7 and Linux kernel 3.10. We present results using the STAMP [31] benchmark suite. STAMP consists of scientific applications from a diversity of fields such as bioinformatics (`Genome`), security (`Intruder`) and machine learning (`Kmeans`). Our analysis makes use of the STAMP applications written in the C language, used as-is and without any modifications. Every application was executed with the *real-world* input-sizes as recommended by the authors [31]. All binaries were compiled from the exact same source code for all three evaluated compiler-based solutions.

[12]Between 20% and 45%, on average, for the evaluated applications.
[13]Average variance across measurements was lower than 0.5% of the mean.
[14]Vectorization was disabled with the `-fno-tree-vectorize`, `-fno-slp-vectorize` and `-fno-vectorize` flags.

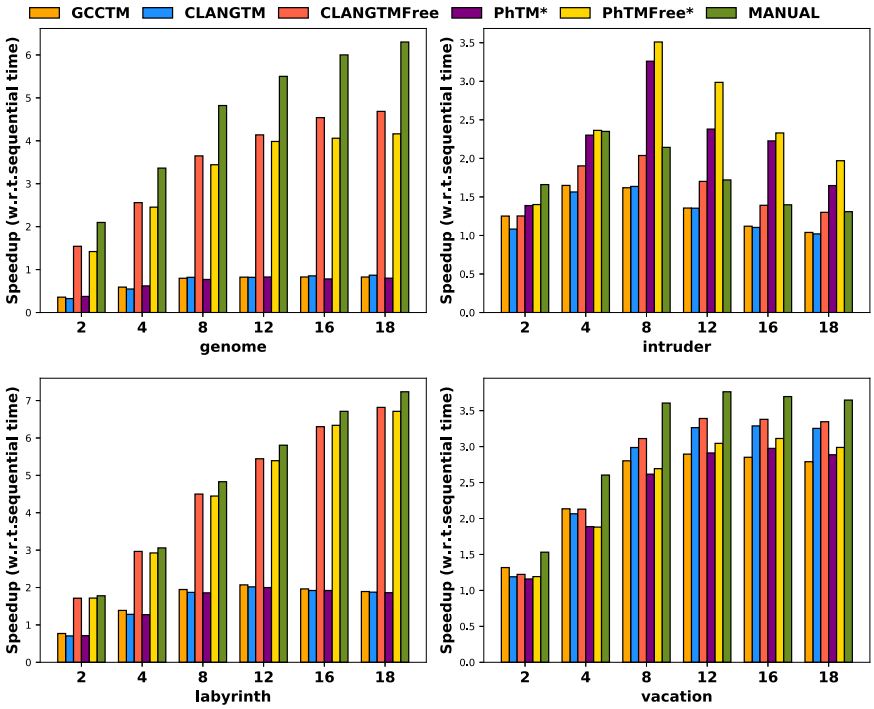


Fig. 5. Speedup results for the STAMP benchmark.

Three[15] applications from STAMP were not used, namely `Kmeans`, `SSCA2` and `Yada`. The first two because they do not achieve significant speedup with software TM [10], and our goal is to show improvements via barrier elision on the software path. `Yada` was the only STAMP application that fails to run with both `GCCTM` and `CLANGTM`. Nonetheless, the four applications in Figure 5 showcase the performance gains obtained by eliding unnecessary barriers.

5.2 Performance Gains Through Barrier Elision

Figure 5 contrasts the performance improvements enabled by our novel annotation mechanism (`CLANGTMFree`) with the manually instrumented code (`MANUAL`) and the transactional code generated by GCC (`GCCTM`). We also show the performance of the phased TM runtime `PhTM*` with the `TMFree` annotation mechanism (`PhTMFree*`), and without it (`PhTM*`). Finally, we present the performance of our transactional support implemented on Clang without the elision pass (`CLANGTM`). Figure 5 shows the speedup (y-axis), relative to the sequential version, achieved in four STAMP applications when executed with 2 to 18 threads (x-axis). Table 1 shows the total number of barriers inserted by each compiler, the percentage reduction (due to the `TMFree` annotation) in `CLANGTM` and `CLANGTMFree` when compared with `GCCTM`, and the number of program lines annotated with `TMFree` for the single-threaded run. These values were measured using the proposed profiling tool and methodology discussed in Section 4. As Figure 5 shows, our proposed annotation (`CLANGTMFree`) mechanism enables `CLANGTM` to scale like `NOrec` does for the manually annotated code (`MANUAL`).

[15]Bayes is no longer used by the research community due to its high execution time variance.

Table 1. Total Number of Inserted Barriers, Percentage Reduction and Code Lines Changed in STAMP Applications

	GCCTM	CLANGTM		CLANGTMFree		
Application	**Total barriers**	**Total barriers**	**% reduction**	**Total Barriers**	**% reduction**	**# lines changed**
Genome	5,909,305,159	5,856,815,828	0.80	285,250,788	95.17	4
Labyrinth	1,909,024,753	1,901,348,025	0.40	6,164,289	99.70	9
Intruder	1,210,415,493	1,185,013,401	2.10	729,883,773	39.70	13
Vacation	1,885,753,418	1,809,386,196	4.00	1,720,921,973	8.70	11

Genome reconstructs a nucleotide sequence from possibly duplicated segments. The reconstruction is divided into three steps. The first and most time consuming step, takes over 90% of the whole execution time [10]. In this step, the only data-structure that is modified is a hash-table used to filter out duplicated short segments. However, the data items themselves are read-only in this step and do not require transactional barriers to be read. These barriers can be easily elided by adding the TMFree annotation to Genome's source code, a change of only four lines of code elides over 95% of the unnecessary barriers.

Similarly, CLANGTMFree elided over 39% of the unnecessary barriers in Intruder, which resulted in a performance improvement of over 23% when compared with GCCTM running with six threads. For Intruder, the performance difference between CLANGTMFree and MANUAL is less than 5% in the 4-thread runs. In order to achieve this performance, only 13 lines of Intruder's code had to be annotated with the TMFree qualifier. Most annotations were made to the data structures used by Intruder to decode packages in its intrusion detection algorithm. The best performance for this application is achieved with PhTMFree* and eight threads of execution, improving performance with respect to the manually annotated code MANUAL by 37%. For both Genome and Intruder, GCC has to add extra barriers since it cannot prove the values are either read-only or transaction local due to this information only being available and known by the programmer.

Labyrinth is an application that finds the shortest-distance path between two points in a maze. The maze is represented as an N-dimensional grid and in Labyrinth is stored as a 3-D array. The routing process can be divided into three steps: expansion, traceback and consolidate. The expansion recursively considers each of the six neighbors of a given source point, taking into account the cost of adding a neighbor to the path. The expansion stops once the destination point is reached. The shortest-distance path is created in the traceback step by traversing the minimum cost path in reverse order, from destination to source. It is important to note that the grid used during both the expansion and traceback steps is a private snapshot of the global grid, taken just before expansion starts. Finally, the consolidate step checks if the path produced during traceback is still available in the global grid. If so, the path is committed, otherwise the path is discarded and the whole process restarts from expansion with an up-to-date snapshot of the global grid. The only grid accesses that need to be transactional are those performed during the consolidate step. Therefore, by annotating only nine lines of code, CLANGTMFree is able to outperform GCCTM by over 3.6× and be within 4% of MANUAL NOrec when 18 threads are used , showing a similar performance to PhTMFree* that presents a speedup of 6.5×.

Vacation was the only application for which CLANGTM was able to elide the barriers without the TMFree annotation. The elided barriers were accesses to captured memory [12], which are dynamically allocated in the heap and are private to the transaction while it is not committed. A profiling analysis using the tool and methodology proposed in Section 4 revealed that less than 10% of the barriers inserted by GCC in Vacation can be elided. Even with such small improvement space, CLANGTM is able to achieve over 20% speedup when compared to GCCTM by eliding barriers to captured memory accesses. Despite annotating 11 lines of code and eliding over 111K

Table 2. Total Number of Commits and Aborts for PhTMFree* and PhTM* with 18 Threads

	PhTM*				PhTMFree*			
	SW		HW		SW		HW	
Application	Commits	Aborts	Commits	Aborts	Commits	Aborts	Commits	Aborts
Genome	61,142.43	964.42	274,116.33	141,854.77	19,417.00	951.04	1,761,236.77	346,213.66
Labyrinth	28.95	0.18	11.80	129.10	28.80	12.01	15.30	117.10
Intruder	329,965.38	157,980.13	10,700,701.50	20,963,293.00	258,708.84	200,343.81	13,022,907.16	26,444,925.33
Vacation	116,030.16	20.60	3,351.40	151,973.90	116,185.46	20.01	1,608.00	111,049.10

barriers, CLANGTMFree only improves CLANGTM's performance by less than 1%. This behavior is also observed with PhTM* and PhTMFree* with maximum speedups of 3.0x and 3.2x, respectively, also improving performance over GCCTM. Honorio et al. [17] achieved similar speedups to CLANGTM using their pragma mechanism. However, due to the limitations discussed in Section 3.2, they could only do so by changing the red-black tree lookup algorithm used by Vacation. The improvements obtained with CLANGTM do not require any source code modification. These results show that identifying the right barriers to elide is central to eliminating overhead and achieving higher performance.

Some of the results presented in this section, specifically those for Genome, could be also achieved by declaring some functions as transactional pure. However, this is not the case of the other three applications; memory allocation would not be performed by the transactional runtime resulting in a memory leak. In addition, purifying[16] functions can be harmful [34], as none of the accesses inside of them are instrumented and thus could violate the semantics of transactional code if used wrongly. The pure attribute– introduced in Section 2.1.1 – is also currently not considered for the C++ TMTS [36]. Nevertheless, we are uncertain if the C++ TM standard should really omit the transactional pure attribute, since if used correctly it could eliminate very significant overhead from the application as this section's results show. In addition, not providing useful features only to prevent programming mistakes is shortsighted. Instead, compilers and runtimes should provide mechanisms to help programmers from making such mistakes and guide barrier elision.

5.3 Mode usage in PhTM*

In this section we show the impact the proposed annotation may have in the performance of phase-based transactional memory systems. For that, we measured the percentage of time spent in each of the execution modes (HW, SW and GCLOCK) for the four STAMP applications considered in the previous section, while also increasing the number of threads from 2 to 18 as shown in Figure 6. For each thread configuration, the left bar corresponds to PhTM* and right one to PhTMFree*. In the per-application description presented next, we also make use of the quantitative data provided by Table 2 for commits and aborts in both systems.

In Genome as the number of threads increases, the percentage of SW mode usage increases for PhTMFree* and PhTM*. However, for PhTM*, this increase is significantly higher, reaching up to 80% of the time in SW which, as shown in the Table 2, executes approximately three times more transactions in SW mode than it does in PhTMFree*. This difference in behavior is due to the fact that the automaton generates a faster return to HW mode (Figure 2, transition (b)), due to the decrease in the transaction duration in SW mode caused by the barriers elision resulting from TMFree. In this case, with PhTMFree*, Genome has a speedup between 1.5× and 4× for increased number of threads, versus a speedup of 4.4× with 18 threads for the same application with CLANGTMFree, whose difference is due to the overhead generated by the PhTM* runtime.

[16]Making functions TM_PURE to elide all barriers in their memory accesses.

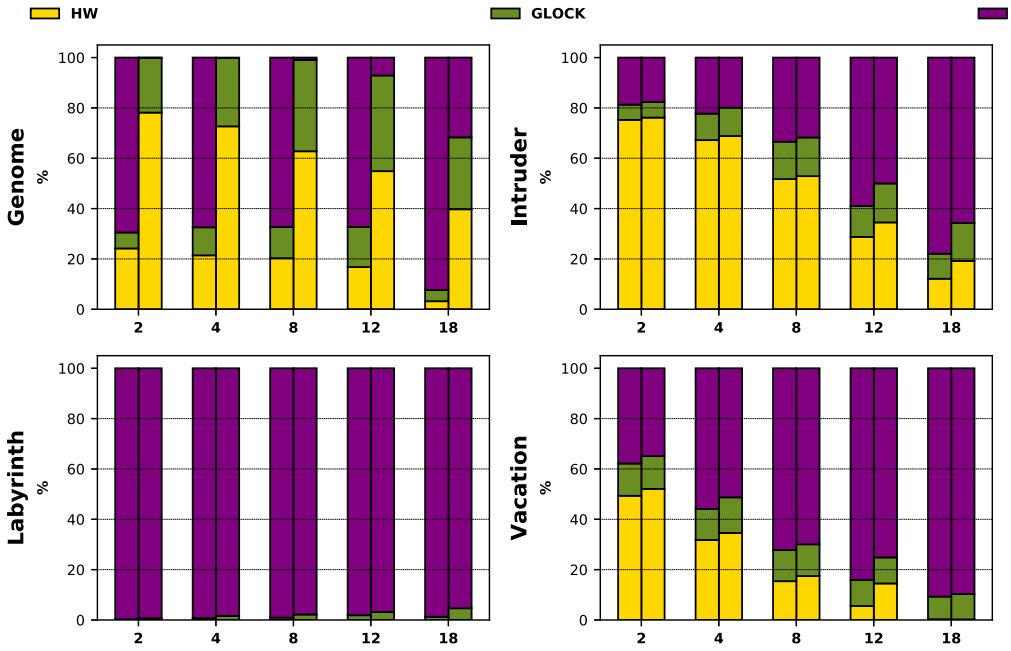


Fig. 6. Mode usage for STAMP benchmark with PhTM* (left-side bar), and PhTMFree* (right-side bar).

Labyrinth executes over 90% in SW mode, which is the reason why the performance gain is similar to the execution with pure STM. The difference in the use of SW mode with PhTMFree* and PhTM* is minimal, as is the number of commits in each mode and, therefore, we can state that the improvement in the performance of this application is due specifically to the decrease in barriers obtained with the TMFree annotation.

For Intruder with PhTMFree*, the usage percentage and the number of transactions executed in SW mode decreases 17% with respect to PhTM*, showing a performance improvement that increases from 1.7× with PhTM* to 2.0× with PhTMFree*, with 18 threads. This application scales up to 8 threads, reaching a maximum speedup of 3.5× with PhTMFree*. Intruder increases SW mode usage from 18% with 2 threads, up to 65% with 18 threads; speedup increases up to 1.20X over PhTM* with no annotation. Notice that this program performs better with PhTMFree* than with MANUAL because of the additional gain provided by the portion of execution in HTM mode.

Vacation has a minimally superior behavior with PhTMFree* compared to PhTM* since the decrease in the number of barriers is only 8.7% (Table 1). Additionally, the number of aborts in HW mode increases 1.36× and the number of commits increases 2.08× with PhTM*.

Overall, we can notice that the time spent in SW mode increases with higher levels of concurrency, but this increase tends to be smaller for the version that uses the annotation (PhTMFree*), showing its importance for PhTM systems as it allows the fastest (HW) mode to be chosen more often.

6 CONCLUSION

By extending the results in [11], this paper presents the first transactional memory support for Clang/LLVM, which can be used as a drop-in replacement of GCC. It also shows how the over-instrumentation problem manifests, as well as its impact on performance when using runtimes that execute fully or partially with STM. More specifically, results with the STAMP suite reveal

that compiler-generated transactional code can be over 7× slower than manually instrumented transactions. To bridge this performance gap, a type qualifier (`TMFree`) was added to the TM language support in Clang/LLVM to enable barrier elision of variables that do not require instrumentation. By using `TMFree` on key points in the application's code, it was possible to achieve scalable performance – both with a STM and a PhasedTM runtime – close to the manual instrumentation, falling behind by only 8% in the worst case. The code annotation was guided by the barrier information provided by a novel memory profiling tool also presented in this article. Together with the profiling tool, `TMFree` demonstrated to be effective in reducing the over-instrumentation problem, and a significant step towards making TM more widely adopted, even for the phased TM runtimes that execute a portion of the transactions in STM. Experimental results with `PhTM*` showed how programs that execute in SW mode can benefit from the `TMFree` annotation and reduce the performance gap between STM and HTM. Moreover, the addition of TM support to the Clang/LLVM compiler framework, the *de facto* standard in compiler and optimization research, aims to encourage further research to resolve other issues on language support for memory transactions.

REFERENCES

[1] 2012. Transactional Memory in GCC, Free Software Foundation. http://gcc.gnu.org/wiki/TransactionalMemory. (2012). Retrieved in 02/Mar/2021.

[2] Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun. 2007. The OpenTM transactional application programming interface. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*. 376–387.

[3] Alexandro Baldassin, Edson Borin, and Guido Araujo. 2015. Performance implications of dynamic memory allocators on transactional memory systems. In *Proceedings of the 20th Symposium on Principles and Practice of Parallel Programming*. 87–96. https://doi.org/10.1145/2688500.2688504

[4] Nathan G. Bronson, Christos Kozyrakis, and Kunle Olukotun. 2009. Feedback-directed barrier optimization in a strongly isolated STM. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 213–225.

[5] Irina Calciu, Justin E. Gottschlich, Tatiana Shpeisman, Gilles A. Pokam, and Maurice Herlihy. 2014. Invyswell: A hybrid transactional memory for Haswell's restricted transactional memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques*. 187–200. https://doi.org/10.1145/2628071.2628086

[6] Brian D. Carlstrom, Austen McDonald, H. Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. 2006. The Atomos transactional programming language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1–13.

[7] Fernando Miguel Carvalho and Joao Cachopo. 2013. Runtime elision of transactional barriers for captured memory. In *Proceedings of the 18th Symposium on Principles and Practice of Parallel Programming*. 303–304. https://doi.org/10.1145/2442516.2442556

[8] Luke Dalessandro, Francois Carouge, Sean White, Yossi Lev, Mark Moir, Michael L. Scott, and Michael F. Spear. 2011. Hybrid NOrec: A case study in the effectiveness of best effort hardware transactional memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. 39–52.

[9] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. 2010. NOrec: Streamlining STM by abolishing ownership records. In *Proceedings of the 15th Symposium on Principles and Practice of Parallel Programming*. 67–78.

[10] Joao P. L. de Carvalho, Guido Araujo, and Alexandro Baldassin. 2017. Revisiting phased transactional memory. In *Proceedings of the International Conference on Supercomputing*. 25:1–25:10. https://doi.org/10.1145/3079079.3079094

[11] Joao P. L. de Carvalho, B. C. Honorio, A. Baldassin, and G. Araujo. 2020. Improving transactional code generation via variable annotation and barrier elision. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 1008–1017. https://doi.org/10.1109/IPDPS47924.2020.00107

[12] Aleksandar Dragojevic, Yang Ni, and Ali-Reza Adl-Tabatabai. 2009. Optimizing transactions for captured memory. In *Proceedings of the 21st Annual ACM Symposium on Parallel Algorithms and Architectures*. 214–222.

[13] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. 2010. Time-based software transactional memory. *IEEE Transactions on Parallel and Distributed Systems* 21, 12 (Dec. 2010), 1793–1807.

[14] Tim Harris, James Larus, and Ravi Rajwar. 2010. *Transactional Memory* (2nd ed.). Morgan & Claypool Publishers. 263 pages.

[15] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. 289–300.

[16] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers.
[17] Bruno C. Honorio, Joao P. L. de Carvalho, and Alexandro Baldassin. 2018. On the efficiency of transactional code generation: A GCC case study. In *2018 Symposium on High Performance Computing Systems (WSCAD)*. 184–190. https://doi.org/10.1109/WSCAD.2018.00037
[18] Susan Horwitz. 1997. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst.* 19, 1 (Jan. 1997), 1–6. https://doi.org/10.1145/239912.239913
[19] Intel Corporation 2009. *Intel Transactional Memory Compiler and Runtime Application Binary Interface* (1.1 ed.). Intel Corporation.
[20] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. 2006. Hybrid transactional memory. In *Proceedings of the 11th Symposium on Principles and Practice of Parallel Programming*. 209–220.
[21] João Paulo Labegalini de Carvalho, Guido Araujo, and Alexandro Baldassin. 2018. The case for phase-based transactional memory. *IEEE Transactions on Parallel and Distributed Systems* 30 (07 2018), 1–1. https://doi.org/10.1109/TPDS.2018.2861712
[22] William Landi and Barbara G. Ryder. 1991. Pointer-induced aliasing: A problem classification. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Citeseer, 93–103.
[23] William Landi and Barbara G. Ryder. 1991. Pointer-induced aliasing: A problem classification. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'91)*. Association for Computing Machinery, New York, NY, USA, 93–103. https://doi.org/10.1145/99583.99599
[24] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization (CGO'04)*. IEEE Computer Society, Washington, DC, USA, 75–. http://dl.acm.org/citation.cfm?id=977395.977673
[25] Chris Lattner and Jacques Pienaar. 2019. MLIR primer: A compiler infrastructure for the end of Moore's law. (2019).
[26] H. Q. Le, G. L. Guthrie, D. E. Williams, M. M. Michael, B. G. Frey, W. J. Starke, C. May, R. Odaira, and T. Nakaike. 2015. Transactional memory support in the IBM POWER8 processor. *IBM Journal of Research and Development* 59, 1 (Jan. 2015), 8:1–8:14. https://doi.org/10.1147/JRD.2014.2380199
[27] Yossi Lev and Maurice Herlihy. 2009. tm_db: A generic debugging library for transactional programs. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*. 136–145.
[28] Yossi Lev, Mark Moir, and Dan Nussbaum. 2007. PhTM: Phased transactional memory. In *Second ACM SIGPLAN Workshop on Transactional Computing*.
[29] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer, and Michael L. Scott. 2006. Lowering the overhead of nonblocking software transactional memory. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. http://www.cs.rochester.edu/research/synchronization/pubs.shtml.
[30] Alexander Matveev and Nir Shavit. 2015. Reduced hardware NOrec: A safe and scalable hybrid transactional memory. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems*. 59–71. https://doi.org/10.1145/2694344.2694393
[31] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*. 35–46.
[32] Takuya Nakaike, Rei Odaira, Matthew Gaudet, Maged M. Michael, and Hisanobu Tomari. 2015. Quantitative comparison of hardware transactional memory for Blue Gene/Q, zEnterprise EC12, intel core, and POWER8. In *Proceedings of the 42nd International Symposium on Computer Architecture*. 144–157. https://doi.org/10.1145/2749469.2750403
[33] Valentin Robert and Xavier Leroy. 2012. A formally-verified alias analysis. In *International Conference on Certified Programs and Proofs*. Springer, 11–26.
[34] Wenjia Ruan, Yujie Liu, and Michael Spear. 2014. STAMP need not be considered harmful. In *Ninth ACM SIGPLAN Workshop on Transactional Computing*.
[35] Wenjia Ruan, Yujie Liu, Chao Wang, and Michael Spear. 2013. On the platform specificity of STM instrumentation mechanisms. In *Proceedings of the International Symposium on Code Generation and Optimization*. 1–10.
[36] ISO/IEC JTC1 SC22. 2015. *Technical Specification for C++ Extensions for Transactional Memory* (n4514 ed.). ISO copyright officer.
[37] Nigel Stephens. 2019. New technologies in the ARM architecture. *ARM Ltd. Linaro Connect Bangkok* (April 2019).
[38] Herb Sutter and James Larus. 2005. Software and the concurrency revolution. *Queue* 3, 7 (Sept. 2005), 54–62.
[39] Ruben Titos-Gil, Manuel E. Acacio, Jose M. Garcia, Tim Harris, Adrian Cristal, Osman Unsal, Ibrahim Hur, and Mateo Valero. 2012. Hardware transactional memory with software-defined conflicts. *ACM Trans. Archit. Code Optim.* 8, 4, Article 31 (Jan. 2012), 20 pages. https://doi.org/10.1145/2086696.2086710
[40] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. 2007. Code generation and optimization for transactional memory constructs in an unmanaged language. In *Proceedings of the International Symposium on Code Generation and Optimization*. 34–48.

[41] Peng Wu, Maged M. Michael, Christoph von Praun, Takuya Nakaike, Rajesh Bordawekar, Harold W. Cain, Calin Cascaval, Siddhartha Chatterjee, Stefanie Chiras, Rui Hou, Mark Mergen, Xiaowei Shen, Michael F. Spear, Hua Yong Wang, and Kun Wang. 2009. Compiler and runtime techniques for software transactional memory optimization. *Concurrency and Computation: Practice and Experience* 21, 1 (Jan. 2009), 7–23. https://doi.org/10.1002/cpe.v21:1

[42] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. 2013. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1145/2503210.2503232

[43] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. 2008. Kicking the tires of software transactional memory: Why the going gets tough. In *Proceedings of the 20th Annual ACM Symposium on Parallel Algorithms and Architectures*. 265–274.

[44] Pantea Zardoshti, Tingzhe Zhou, Pavithra Balaji, Michael L. Scott, and Michael Spear. 2019. Simplifying transactional memory support in C++. 16, 3, Article 25 (July 2019), 24 pages. https://doi.org/10.1145/3328796

Received November 2021; revised February 2022; accepted April 2022