Scientific Report for STSM project "T-Rex: A Dynamic Race Detection Tool for C/C++ Transactional Memory Applications"

March 13, 2012

1 Purpose of the STSM

The dominance of multi-core processors has made concurrent programming essential to achieve peak performance from modern systems. Unfortunately, despite the performance benefit, parallel programming introduces high software complexity and is prone to synchronization bugs such as data races, atomicity violations, deadlock and livelock.

Transactional Memory (TM) is one of the suitable models that addresses ease of programmability of parallel programs while keeping up with performance expectations of multi-core processors. Using TM, programmers mark sections of code they intend to execute atomically, rather than protecting shared memory locations with locks, and need not worry about deadlocks. There have been significant efforts to develop TM systems, hardware (HTM) [3, 6] and software (STM) [4, 10, 2, 7] and compilers with TM support [8, 1]. However, there is still lack of software development tools and integrated environments that help programmers debug and analyze TM applications, such as race detection tools.

The purpose of this short term scientific mission (STSM) was to tackle the problem of identifying race conditions in C/C++ TM programs running on STM systems, independently of the particular STM implementation used by the programmer. Data race detection for lock-based applications has been intensively studied in the literature but, to the best of our knowledge, this work is the first to explicitely address race detection for real C/C++ TM applications. Informally, a data race is defined as a condition where multiple threads access a shared memory location without proper synchronization and there is at least one write among the memory operations. Data races are difficult to diagnose and reproduce because they often manifest under certain, likely to be very rare, thread interleavings. Although the definition of data race is orthogonal to the synchronization mechanism, critical sections protected by locks and transactions are semantically different and present distinct characteristics and requirements, thus race detection tools used for lock-based applications cannot be directly extended to transactional memory. With TM, a transaction either executes completely and atomically or should appear as if it were never executed. This means that transactions' effects should be permanently visible, and thus can generate data races, only once a transaction has successfully committed.

We propose *T-Rex*, a novel dynamic race detection tool for C/C++ transactional memory applications. Our tool records read and write accesses to shared memory locations, both inside and outside transactions, into per-thread metadata and then detects races at synchronization points. Moreover, *T-Rex* is agnostic of thread interleavings, as it does not depend on the particular threads' execution ordering to determine data races. *T-Rex* is transparent to the underlying TM systems and track memory accesses through Pin, a dynamic binary instrumentation tool.

We evaluate *T-Rex* with a widely-used STM design, TL2[4], a lazy-acquiring, lazy-updating TM system, using applications from the STAMP benchmark suite on an 8-core Intel Nehalem server.

2 Description of the work carried out during the STSM

This section describes the formal definition of a race condition for transactional memory applications and presents the implementation of *T-Rex*.

2.1 Preliminaries

We propose the following definition of a data race:

Definition 1. A data race among $N > 1$ threads $\{T_1, ..., T_N\}$ that access the same shared memory location occurs when:

i. at least one access is a write, and

ii. at least one access is unprotected.

This definition is agnostic of thread interleaving and relies on the more intuitive idea that two accesses to a shared memory location without the proper synchronization mechanism would probably result in a data race, even if they do not generate a race in that particular execution.

Let's define W_i^u and R_i^u as the sets of memory locations written and read by thread T_i inside unprotected regions, respectively, and W_i^p and R_i^p as the sets of memory locations written or read by thread T_i within protected regions, respectively. Then, a race occurs between thread T_i and T_j iff at least one of the following is true:

$$c_i^1(j) = W_i^u \cap \{R_j^u \cup W_j^u \cup R_j^p \cup W_j^p\} \neq \oslash \quad (1)$$

$$c_i^2(j) = R_i^u \cap \{W_j^u \cup W_j^p\} \neq \oslash \quad (2)$$

$$c_i^3(j) = W_i^p \cap \{R_j^u \cup W_j^u\} \neq \oslash \quad (3)$$

$$c_i^4(j) = R_i^p \cap \{W_j^u\} \neq \oslash \quad (4)$$

We denote with $\mathbb{S}_i(j)$ the set of races that thread T_i has with thread T_j; $\mathbb{S}_i(j)$ is the sum of (1), (2), (3), and (4):

$$\mathbb{S}_i(j) = c_i^1(j) \cup c_i^2(j) \cup c_i^3(j) \cup c_i^4(j) \quad (5)$$

It follows that thread T_i has data races with thread T_j iff $\mathbb{S}_i(j) \neq \oslash$.

For $N > 2$ concurrent threads, a thread T_i may have races with multiple threads at the same time. This would require computing a combinatory number of sets of races, which would be impractical from a performance point of view. However, the following theorem holds true:

Theorem 1. *The set of races that thread T_i has with threads $\{T_j, T_k\}$ is equivalent to the union of the set of races that thread T_i has with thread T_j and with T_k:*

$$\mathbb{S}_i(j,k) = \mathbb{S}_i(j) \cup \mathbb{S}_i(k)$$

Proof. This theorem can be proved using the commutativity and distributivity properties of set unions and intersections. Let's consider $N = 3$ threads, namely T_i, T_j, and T_k, with $i \neq j \neq k$. Then conditions (1)-(4) can be re-written as:

$$\begin{aligned} c_i^1(j,k) = W_i^u \cap \{\{R_j^u \cup R_k^u\} \cup \{W_j^u \cup W_k^u\} \\ \cup \{R_j^p \cup R_k^p\} \cup \{W_j^p \cup W_k^p\}\} \end{aligned} \tag{6}$$

$$c_i^2(j,k) = R_i^u \cap \{\{W_j^u \cup W_k^u\} \cup \{W_j^p \cup W_k^p\}\} \tag{7}$$

$$c_i^3(j,k) = W_i^p \cap \{\{R_j^u \cup R_k^u\} \cup \{W_j^u \cup W_k^u\}\} \tag{8}$$

$$c_i^4(j,k) = R_i^p \cap \{W_j^u \cup W_k^u\} \tag{9}$$

By applying the distributivity property to, for instance, (6) and re-grouping the terms, we obtain:

$$\begin{aligned} c_i^1(j,k) = W_i^u \cap \{R_j^u \cup W_j^u \cup R_j^p \cup W_j^p\} \cup \\ W_i^u \cap \{R_k^u \cup W_k^u \cup R_k^p \cup W_k^p\} = c_i^1(j) \ \cup \ c_i^1(k) \end{aligned}$$

The same principle can be applied to (7)-(9) with the same results. Putting everything together, we obtain the thesis. □

The equality in Theorem 1 can be also read in the reverse order: if $\mathbb{S}_i(j)$ and $\mathbb{S}_i(k)$ are known, $\mathbb{S}_i(j,k)$ can be simply obtained by summing the known sets. This is of practical importance for *T-Rex*, as $\mathbb{S}_i(j)$ and $\mathbb{S}_i(k)$ can be determined independently and in parallel. Theorem 1 can be generalized to $N > 3$ through the following corollary:

Corollary 1. *The set of races that thread T_i has with threads $\{T_{j1}1, T_{j2}, \ldots, T_{jm}\}$ is equivalent to the union of the set of races that thread T_i has with thread T_{j1}, T_{j2},...,T_{jm}:*

$$\mathbb{S}_i(j_1, j_2, \ldots, j_m) = \bigcup_{k=1}^{m} \mathbb{S}_i(j_k)$$

Proof. The corollary can be proved using the same technique applied to the proof of Theorem 1. For brevity, we omit the formal proof. □

Using Corollary 1, the definition of set of races (5) can easily be extended to $N > 2$ threads:

Definition 2. Given N concurrent threads, the set of races, $\mathbb{S}_i$, of thread T_i with threads T_j, $\forall j \neq i$ is:

$$\mathbb{S}_i \quad = \quad \bigcup_{i \neq j} \mathbb{S}_i(j) \tag{10}$$

From the practical point of view, however, this definition still requires computing $\mathbb{S}_i(j)$ for each $i, j = 1..N, i \neq j$, where N is the number of concurrent threads. The total number of sets of races to compute for N threads is $N \times (N-1)$: this procedure is time consuming, especially for large values of N. The following theorem reduces the complexity of computing $\mathbb{S}_i$ for $i = 1..N$.

Theorem 2. *The set of races that thread T_i has with thread T_j is equivalent to the set of races that thread T_j has with thread T_i:*

$$\mathbb{S}_i(j) = \mathbb{S}_j(i)$$

Proof. The theorem can be proved by applying the commutativity and distributivity properties of set unions and intersection and by properly grouping terms. □

Theorem 2 states that $\mathbb{S}_j(i)$ can be easily obtained if $\mathbb{S}_i(j)$ is known. The theorem reduces the number of sets of races to be computed to $N \times (N-1)/2$.

Definition 1 is very generic and not all unprotected accesses to shared memory locations necessarily generate data races. There might be cases in which the programmer knows that a data race can never occur, e.g., because there is an explicit (`barrier()`) or implicit synchronization, or sequential semantic is guaranteed by the underlying memory model (i.e., `atomic` variables in C++ 0x). Moreover, the following corollary holds true:

Corollary 2. *Serial sections of an application do not generate data races.*

Proof. During serial sections of an application the number of concurrent threads is $N = 1$. From Definition 1, it follows that there cannot be races with $N = 1$ active threads. □

This corollary implies that a race detection system need not record those accesses (reducing runtime overhead) while still maintaining precision. It also allows the intuitive programming paradigm of initializing global variables in the main thread or writing the final results of an application without using synchronization primitives.

Race detection systems may provide false positives — potential races that are, indeed, not races. Examples of false positives include accesses to `atomic` variables in C++ 0x unprotected regions or accesses to shared memory locations protected by implicit synchronization mechanism (e.g., reading from a common file or mechanism provided by the programmer). In these cases, the race detection tool reports races and then it is up to the programmer to distinguish which ones are real and which ones are false positives. Obviously, if the number of false positives is very high, this task becomes daunting. In some cases, however, the number of false positives can be drastically reduced by identifying explicit synchronization points. The next Theorem formalizes this concept:

Theorem 3. *Accesses to shared memory locations across global synchronization primitives do not generate a race.*

Proof. The Theorem can be demonstrated by contradiction. Let's assume that two application threads T_i and T_j (the extension to more than 2 threads is trivial) reach a global synchronization point at time $t = t_b$. Let's also assume that a thread T_i accesses a shared memory location l at time $t = t_1 < t_b$ while

another thread T_j accesses the same memory location at time $t = t_2 > t_b$ and that this accesses generate a race across the barrier. By Definition 1, there is a possible execution in which T_i and T_j access concurrently l, hence $t_1 = t_2$. However, by definition a global synchronization primitive (such as barriers, join, etc.) is a point in the program that has to be reached by all threads before proceeding to the next section. Hence T_i and T_j are either both before the synchronization point ($t_1, t_2 < t_b$) or both after ($t_1, t_2 > t_b$), which contradicts the initial hypothesis ($t_1 < t_b$ and $t_2 > t_b$), thus T_i and T_j cannot possibly generate a race across barriers when accessing l. □

2.2 Design and Implementation

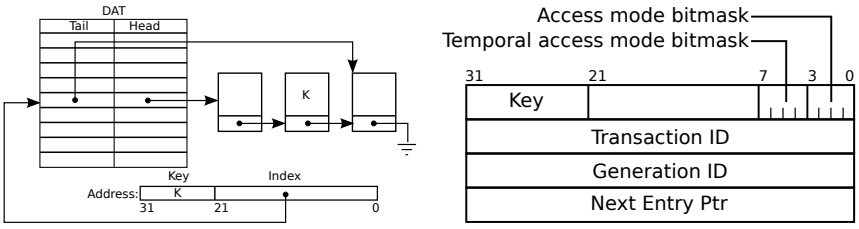
2.2.1 Threads Data Access Table

To detect possible data races agnostically of thread interleaving, all individual protected and unprotected accesses to shared memory locations and the access mode (read/write, protected/unprotected) have to be recorded. *T-Rex* stores each access into a per-thread data access table (DAT), shown in Figure 1a. The per-thread DAT is implemented as a hash table and the hash function uses the last 22 bits of the memory address (bits [21:0]) to index the table. Addresses that map to the same hash table bucket (aliases) are stored in a linked list, thus all individual accesses are precisely stored and there are no false negatives. However, because of memory and architectural constraints, we detect data races at word level (4 bytes), hence false positives are possible if two threads access, for example, two disjoint portions of the same word. Notice that hardware-level race detection tools usually detect races at cache line granularity (64-128 bytes), which may cause a considerable number of false positives. Gupta et al. [5] conclude that then number of false positives would greatly be reduced if they could use word-level race detection. However, a realistic multiprocessor system cannot be expected to have a 4-byte cache line.

Figure 1b shows the structure of each entry in the per-thread DAT. The first word stores the upper part of the memory address (bits [31:22]) and it is used to disambiguate memory addresses that map to the same hash table bucket. Bits [3:0] (*access mode bitmask*) store the access modes (protected/unprotected, read/write) the location has been accessed by the thread: this bitmask is cross checked with the other threads' to determine possible data races.

2.2.2 Binary instrumentation Framework

Dynamic race detection implies tracking read and write accesses to shared memory locations. For C/C++ applications, the most effective solution to trace read/write accesses would be to have a compiler instrumenting shared memory accesses and the race detection tool detecting whether there is a possible race. Unfortunately, currently, compilers for TM applications only allow programmers to automatic instrument transactional accesses. None of them, to the best of our knowledge, instruments non-transactional accesses or provides hooks for race



(a) Data Access Table (b) Data Access Entry

Figure 1: *T-Rex* bookkeeping data structures.

detection tools. Manual instrumentation, on the other hand, is only possible for simple applications, such as micro-benchmarks, and is prohibitive for large, real applications such as the ones tested in this work. We, thus, implemented *T-Rex* on top of Pin [9], a dynamic instrumentation tool that allows programmers to instrument transactional and non-transactional read and write accesses, as well as functions' entry and exit points. Pin enables users to dynamically modify binary applications on the fly, with no static annotation inserted by the programmer and with no need of re-compiling/re-linking applications, to call custom routines before or after the instrumented instruction is executed. The drawback of dynamic binary instrumentation is the potential runtime overhead introduced. We believe that TM compiler will eventually provide functionalities to instrument shared memory accesses that may generate races. Despite the instrumentation overhead, our results show that *T-Rex* overall overhead is in the same order of state-of-the-art race detection tools.

In oder to distinguish transaction (protected) from non-transactional (unprotected) code, we instrument `TM_BEGIN()` and `TM_END()`: In our implementation, we add a call to `PIN_TM_BEGIN()` before a `TM_BEGIN()` is executed. `PIN_TM_BEGIN()` sets a flag (`is_in_cs`) indicating that the thread has now entered a protected section (transaction). A commit stage, `TM_END()`, the STM library checks whether there are unresolved conflicts and, if not, the transaction commits. This is the moment when all the memory locations modified by the transaction become visible to the other threads, hence, this is the moment when races may eventually occur. `PIN_TM_END()` is invoked at the end of `TM_END()` and marks the memory locations modified by the transaction as permanent. The `is_in_cs` is also cleared. Notice that, on abort, `TM_END()` does not end (the transaction restarts from the beginning), thus, `PIN_TM_END()` is not invoked.

2.2.3 Unprotected Memory Accesses

While transactional memory accesses are annotated, either by the programmer or by the TM compiler, non-transactional shared memory accesses are not annotated there is no easy way to instrument them for C/C++ applications. We use Pin to instrument each non-transactional memory access, though the overhead of such instrumentation is usually larger than compiler-assisted or manual instrumentation. In order to reduce runtime overhead and false positives, we

only instrumented unprotected read and write accesses to shared memory locations performed within the application's code. We discard libraries' memory accesses because they are not under the control of the programmer; moreover, we discard per-thread memory accesses (i.e., variables on the stack), as they do not usually generate races.

Unprotected accesses directly modify the access mode bitmask (bits [3:0]). At every unprotected access, Pin executes either `PIN_READ()` or `PIN_WRITE()` and inserts/updates the entry corresponding to the memory location in the thread's DAT. `PIN_READ()` and `PIN_WRITE()` set the unprotected read (bit [1]) or unprotected write (bit[0]) bit, respectively. Both `PIN_READ()` and `PIN_WRITE()` first look for the memory location address into the threads' DAT and, if found, updates the access mode bitmask. If the thread has never accessed that particular memory location, a new entry is added and the access mode bitmask is initialized with the current access mode. For performance reason, we store both `head` and `tail` of each bucket list: the last element of a bucket list (`tail`) is returned if the required address is not found, making inserting a new item easier.

2.2.4 Protected Memory Accesses

Managing protected accesses to shared memory locations is, instead, more complicated: the semantic of transactional memory implies that modified memory locations are permanently visible to other threads only once the transaction has committed. On the other hand, instrumenting protected accesses is simpler because transactional memory applications are already annotated, either by the programmer or by the TM compiler, with statements that mark read and write accesses to shared memory locations within transaction (`STM_READ()` and `STM_WRITE()`, respectively).

One key design point of *T-Rex* is the STM implementation transparency. This means that *T-Rex* cannot rely on the particular STM implementation and data structures (e.g., the read- and write-set) or any other assumption only valid for a specific STM library. A possible implementation consists of using shadow temporal data structures to record transactional read and write accesses. Once the transaction commits and the memory accesses become permanent, the values in the temporal structures are copied to the thread's DAT. From the performance point of view, however, keeping separate data structures may introduce considerable overhead (memory copy) at commit phase. We, instead, implemented a commit zero-copy algorithm to keep track of transactional accesses. Bits [7:4] in Figure 1b store a temporal access mode bitmask used during transaction execution: We use a *transaction ID* (second word in Figure 1b) to identify memory locations already accessed by the current transaction from those that have never been accessed in the scope of the current transaction. On transactional read/write access, the thread's DAT is searched and, if the memory address is already present in the table, its transaction ID is compared to the current transaction ID. If the entry's transaction ID is smaller than the current transaction ID, this is the first attempt to access that location

transactionally and *T-Rex* copies the access mode bitmask (bits [3:0]) to the temporal access mode bitmask (bits [7:4]), updates the access mode bitmask and the entry's transaction ID, and records the DAT entry. From that moment on, every other transactional operation to the same memory location in the scope of the current transaction directly updates the access mode bitmask (bit [3] or bit [2] for `STM_READ()` and `STM_WRITE()`, respectively). If the memory location is not found in the thread's DAT, a new entry is added and its bitmask and transaction ID are initialized with the current access mode and transaction ID.

If the transaction commits, no further updates of the thread's DAT are required (zero-copy on commit). On aborts, the access mode bitmasks of the locations accessed during the transaction must be rolled back to their original values (stored in the temporal access mode bitmasks). The memory locations that have to be rolled back are stored in the list of individual memory locations accessed by the current transaction. Moreover, memory locations added by the current transaction (temporal access mode bitmask is 0) are removed from the thread's DAT. With our scheme, aborting a transaction is more expensive than committing. On the other hand (see Section 3), the number of commits is, on average, orders of magnitude higher than the number of aborts, thus we generally have a net gain.

2.2.5 Race detection

According to Definition 1, a race occurs iff at least one of the conditions (1)-(4) is true. In our implementation, for each thread T_i, the set R_i^p, W_i^p, R_i^u, and W_i^u are determined by the access mode bitmask in each entry of the thread T_i's DAT (DAT_i). For example, if the entry for the location X stores the access mode bitmask 0110, then $X \in W_i^p$ and $X \in R_i^u$, hence thread T_i can have a race on X with any thread T_j, with $j \neq i$, if $X \in \{W_j^u \cup R_j^u\}$ (condition (3)) or $X \in \{W_j^u \cup W_j^p\}$ (condition (2)).

We can express conditions (1)-(4) in terms of bit operations and detect races through a logic function determined with Karnaugh maps techniques. Let's define $B_i(X)$ the access mode bitmask of thread T_i for location X ($B_i(X) = 0x0$ if $X \notin DAT_i$). Then, for threads T_i and T_j, conditions (1)-(4) can be expressed as:

$$(1) \Rightarrow (B_i(X) \wedge 0x1) \wedge B_j(X) \tag{11}$$

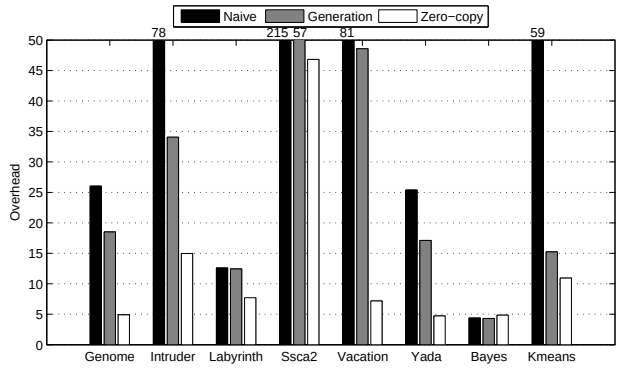
$$(2) \Rightarrow (B_i(X) \wedge 0x2) \wedge (B_j(X) \wedge 0x5) \tag{12}$$

$$(3) \Rightarrow (B_i(X) \wedge 0x4) \wedge (B_j(X) \wedge 0x3) \tag{13}$$

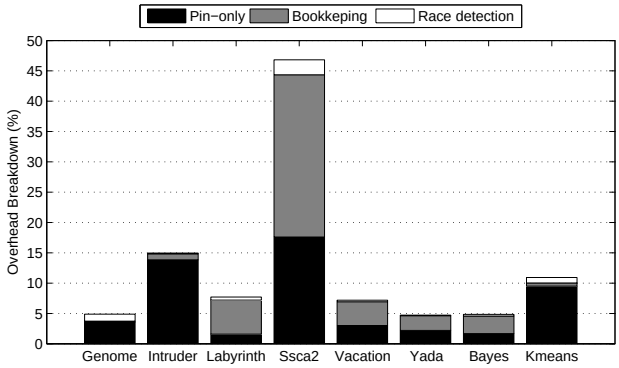
$$(4) \Rightarrow (B_i(X) \wedge 0x8) \wedge (B_j(X) \wedge 0x1) \tag{14}$$

for each $X \in DAT_i$ and the number of races between T_i and T_j on location X can be computed as the sum of the hamming weights of the bitmasks computed in (11)-(14).

By Theorem 1 and Theorem 2, $\forall i, j, k : i \neq j \neq k$, $\mathcal{S}_j(i)$ and $\mathcal{S}_k(i)$ can be independently computed by threads T_j and T_k and $\mathcal{S}_i(j, k)$ can be obtained by



(a) *T-Rex* runtime overhead.



(b) *T-Rex* runtime overhead breakdown.

Figure 2: *T-Rex* overall overhead and overhead breakdown for STAMP applications.

summing these two sets. $\mathcal{S}_j(i)$ and $\mathcal{S}_k(i)$ can be computed in parallel because each thread's DAT is disjoint from the others and race detection only requires reading the threads' DATs (no need to synchronize accesses to DATs). We, thus, implemented a parallel race detection algorithm in which each thread T_i detects races with thread T_j, with $i \neq j$. Moreover, by Theorem 2, we only need to compute the set for one thread (e.g., T_i) and replicate it for the other thread (e.g., T_j). It follows that thread T_i only needs to determine $\mathcal{S}_i(j)$, $\forall j > i$.

3 Description of the main results obtained

This section evaluates the performance of *T-Rex* over native execution of STAMP applications. STAMP applications are widely used to evaluate TM systems and

present different characteristics, such as large/short transactions, large/small read- and write-sets, high/low conflict rate, etc. We evaluated our dynamic race detection tool with applications on a Intel Nehalem system (8 cores, 16GB of RAM); STAMP applications are compiled with gcc 4.4.5 with optimization `O3` for 32-bit architectures. We use TL2 as underlying STM system, also compiled with gcc 4.4.5 and optimization `O3` for 32-bit architectures.

3.1 Overhead analysis

Although race detection tools are not intended to be run in production but only in development and debugging sessions, a low runtime overhead facilitates race analysis and increases productivity. Figure 2a shows the overhead of our race detection tool over the native execution of STAMP applications running with the TL2 STM runtime library. We report the overhead of three different implementations: *Naïve* uses our race detection algorithm but deallocates/allocates threads' DATs at each global synchronization point, after performing race detection, and uses shadow data structures to store temporal transactional read and write accesses that are copied back to the thread's DAT after successful commits. Temporal data structures are allocated at the beginning of a transaction and deallocated at commit phase and on abort. *Generation* employs generation across global synchronization points (no need to deallocate/allocate threads' DATs) but still uses shadow data structures for transactional accesses. To the contrary of the previous case, temporal data structures are not deallocated at the end of transactions but invalidated through a specific generation ID. Finally, *Zero-copy* uses both our optimizations: generation across global synchronization points and zero-copy commit phase.

By comparing versions *Naïve*, *Generation* and *Zero-copy* we can perceive the effects of each optimization. The use of generations across global synchronization points considerably improve performance for those applications (*Genome*, *Kmeans* and *SSCA2*) that use strict synchronization through barriers or fork/join (see Table 1). Moreover, version *Generation* does not deallocate/allocate temporal data structures for transactional accesses, hence applications with a large number of commits (*Intruder*, *SSCA2* and *Kmeans*) also report large performance improvements. Including our zero-copy on commit optimization (version *Zero-copy*), instead, generally improves performance for all the applications. Our zero-copy on commit technique removes the memory copy overhead of moving transactional accesses from the temporal data structure to the threads' DATs, thus applications with a large number of commits, such as *Kmeans*, *Intruder* and *SSCA2*, and/or large read- and write-sets, such as *Vacation* and *Yada*, are the ones that benefits the most from this optimization. As we can see from Figure 2a, the *T-Rex* overhead (*Zero-copy*) is generally below 10x. This value is comparable to state-of-the-art dynamic lock-based race detection tools despite the fact that race detection for TM applications requires extra work to keep track of temporal transactional accesses, make transactional accesses visible to other threads at commit phase and handle aborts.

In some case, such as *SSAC2* and *Intruder*, the runtime overhead is partic-

App.	Commits	Aborts	Syn.	Protected Accesses		Unprotected Accesses	
				Rd	Wr	Rd	Wr
Intruder	23428K	302K	1	559.4	28.9	180.9	15.4
Ssca2	22362K	0.01K	47	22.4	44.7	1,204.4	163.2
Kmeans	8825K	2K	302	2.2	2.1	173.4	0.1
Vacation	4194K	9K	1	1,199.9	22.8	133.8	62.7
Genome	2489K	2K	258	90.2	2.2	9.2	0.1
Yada	2415K	235K	1	141.9	39.2	4.8	0.0
Bayes	2K	0.1K	4	0.1	0.0	0.0	0.0
Labyrinth	1K	0.1K	1	0.2	0.2	0.0	0.0

Table 1: STAMP applications' characteristics. Number of protected/unprotected read and write accesses are reported as millions.

ularly high. In order to identify the reasons of this large overhead, we analyzed the overhead breakdown (Figure 2b) and discovered that, for these benchmarks, the pure Pin instrumentation overhead (no bookkeeping and no race detection) is already very large. The Pin overhead is 9.5x, 13.8x and 17.5x for *Kmeans*, *Intruder* and *SSCA2*, respectively. Other applications, however, show a much lower instrumentation overhead: For *Labyrinth*, for example, the instrumentation overhead is 1.6x. We further examined the reasons why we experience large overhead for *SSAC2* and *Intruder*: Table 1 reports the number of commits, aborts, synchronization points, and protected/unprotected read and write accesses. As we can see, there is a correlation between the number of commits and the Pin overhead: applications with a large number of commits present high Pin overhead caused by the frequent library calls.

From Figure 2b and Table 1 we can also derive conclusions about the bookkeeping, the read and write instrumentation and the race detection overheads. In the graph, bookkeeping also includes the Pin instrumentation overhead of detecting transactional and non-transactional shared memory location accesses. Applications with large numbers of memory accesses, such as *SSCA2* and *Vacation*, show larger bookkeeping overhead. On the other hand, *Intruder* and *Kmeans* present a lower number of memory accesses, thus their bookkeeping overhead is small. This explains why, although *Intruder* and *SSCA2* show a comparable number of commits (therefore a similar instrumentation overhead), the overall overhead of *SSCA2* is larger than that of *Intruder*.

Finally, Figure 2b shows that the pure overhead of our race detection algorithm is marginal with respect to the instrumentation and bookkeeping overhead. Overall, the overhead of race detection is lower than 5% of the overall overhead for all applications, with only *SSCA2* and *Kmeans* between 5% and 10%, and *Genome* above 20%. These three applications are the only ones that frequently use global synchronization primitives, thus *T-Rex* performs race detection several times. As we will see in the following, however, performing race detection at synchronization points considerably reduces the number of false positives.

App.	# Races Detected	# Additional Races w/ bug injection	# Additional Races w/o RDSP
Intruder	1422547	7723*	0
Ssca2	90	3	NA
Kmeans	0	3546	151860
Vacation	0	758354	0
Genome	0	1706	20
Yada	0	6	0
Bayes	260	3	47
Labyrinth	0	7	0

Table 2: Number of detected data races for STAMP applications for 1) the original version, 2) a version with synthetic bugs injected, and 3) without race detection at synchronization points (RCSP). **Intruder* crashed because of the injected bug.

4 Future collaboration with host institution

Both participant sides (BSC and Koc University) are currently discussing the opportunity of publishing a joint paper containing the results of this STSM. The paper will target a high-profile conference like OOPSLA.

5 Confirmation by the host institution of the successful execution of the STSM

The Hosts Serdar Tasiran from the Koc University confirm that Gokcen Kestor achieved all the targets that we defined for this collaboration with distinction. She designed a race checker tool for C/C++ Transactional Memory applications. In fact, this race checker tool is the first attempt in order to detect races for complexed applications that use Transactional Memory as a programming models, and this tool will also enable the researchers to debug their transactional applications without introducing undesirable slowdown.

References

[1] D. Christie, J. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, and E. Riviere. Dresden TM Compiler (DTMC). In *Proc. of the 5th ACM European Conference on Computer Systems*, 2010.

[2] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by abolishing ownership records. In *Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Jan. 2010.

[3] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, 2009.

[4] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[5] S. Gupta, F. Sultan, S. Cadambi, F. Ivancic, and M. Rotteler. Using hardware transactional memory for data race detection. In *Proc. 23rd International Parallel and Distributed Processing Symposium*, 2009.

[6] Ruud Haring. The Blue Gene/Q compute chip. In *The 23rd Symposium on High Performance Chips (Hot Chips)*, 2011.

[7] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory.* Morgan & Claypool Publishers, 2nd edition, 2010.

[8] Intel Corporation. Intel transactional memory compiler and runtime application binary interface, May 2009. Document No. 318523-002US, revision 1.1.

[9] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 190–200, 2005.

[10] T. Riegel, C. Fetzer, and P. Felber. Time-based Transactional Memory with scalable time bases. In *19th ACM Symp. on Parallelism in Algorithms and Architectures*, 2007.