WormBench - A Configurable Workload for Evaluating Transactional Memory Systems

Ferad Zyulkyarov*
Barcelona Supercomputing Center
ferad.zyulkyarov@bsc.es

Sanja Cvijic†
Belgrade University
scvijic@etf.bg.ac.yu

Osman Unsal
Barcelona Supercomputing Center
osman.unsal@bsc.es

Adrian Cristal
Barcelona Supercomputing Center
adrian.cristal@bsc.es

Eduard Ayguade*
Barcelona Supercomputing Center
eduard.ayguade@bsc.es

Tim Harris
Microsoft Research UK
tharris@microsoft.com

Mateo Valero*
Barcelona Supercomputing Center
mateo.valero@bsc.es

ABSTRACT

Transactional Memory (TM) is a promising new technology that makes it possible to ease writing multi-threaded applications. Many different TM implementations exist, unfortunately most of those TM systems are currently evaluated by using workloads that are (1) tightly coupled to the interface of a particular TM implementation, (2) are small and lack to capture the common concurrency problems that exist in real multi-threaded applications and also (3) fail to evaluate the overall behavior of the Transactional Memory considering the complete software stack.

WormBench is parameterized workload designed from the ground up to evaluate TM systems in terms of robustness and performance. Its goal is to provide an unified solution to the problems stated above (1, 2, 3). The critical sections in the code are marked with the atomic statements and thus proving a framework to test the compiler's ability to translate them properly and efficiently into the appropriate TM system interface. Its design considers the common synchronization problems that exist in TM multi-threaded applications. The behavior of WormBench can be changed by using run configurations which provide the ability to reproduce a runtime behavior observed in a typical multi-threaded application or a behavior that stresses a particular aspect of the TM system such as abort handling. In this paper, we analyze the transactional characteristics of WormBench by studying different run configurations and demonstrate how WormBench can be configured to model the transactional behavior of an application from the STAMP benchmark suite.

*This author is also affiliated with the Dept. of Computer Architecture at Universitat Politècnica de Catalunya.
†This work was developed while the author was an intern at Barcelona Supercomputing Center.

MEDEA October 26, 2008 Toronto, Canada

1. INTRODUCTION

The emerging era of Chip-Multiprocessors (CMP) pushed the research community to seek for new techniques to make it easier for application and library developers to develop scalable and efficient multi-threaded applications. Transactional Memory (TM) is an optimistic concurrency control mechanism first proposed by Herlihy and Moss [9] that promises to provide a better solution for the existing concurrency control scenarios in the multi-threaded applications with shared global state. Its simple programming language interface, abstracts away the complexity of writing multi-threaded applications. Instead of tracking each shared data and enforcing serial access by using locks, using atomic blocks, the programmer only identifies the code segments that must be executed atomically and leave the underlying infrastructure handle synchronization. Its non-blocking nature prevents lock induced deadlocks. Typical implementations aim to scale comparably with fine grain lockings. Based on the implementation, TM exists in two flavors - Hardware (HTM) [3, 22] and Software (STM) [10, 5, 17]. HTM is implemented at the micro-architectural level, it is fast but limited in time and space. STM is implemented as a runtime library, it is unbounded both in time and space but because of the incurred overhead is slow. Also, there exist hybrid solutions that either try to virtualize HTMs [4, 16] or accelerate STMs [18, 13].

Typically, the performance characteristics of the currently proposed TM systems are evaluated by using a small number of workloads. Mainly these are small application kernels - μbenchmarks. Recently, more complex TM applications were developed for benchmarking by either transactifying from lock-based versions (STMBench7 [7], applications from the SPLASH-2 benchmark suite [21]) or by writing TM applications from scratch (STAMP [12] and Haskell STM benchmark suite [15]). The μbenchmarks perform simple type of operations such as lookup, insertion and deletion, on simple data structures like a linked list and hash tables. They are suitable for evaluating specific low level implementation details of the TM systems, but they are not suitable to evaluate how the proposed TM system fits with the other components in the system. The transactional applications are more descriptive compared to the μbenchmarks as

they perform computations in and outside the transactions. STAMP has longer running transactions with small objects and is targeted for evaluating both HTM and STM. STMBench7 has long running transactions with large objects and is designed for evaluating STM systems. SPLASH-2 has inherently parallel code with small critical sections that guard small objects. It has been used to evaluate the following HTM systems [4, 22]. Haskell STM benchmark is set of applications implemented using the language level constructs in the Haskell programming language and do not expose any implementation details of the underlying STM system. However their implementation rely on Haskell's type system and wouldn't be representative if ported to imperative language. The other discussed TM applications are implemented by exposing the implementation details of the TM system that they are targeted for. This makes it difficult to compare different TM systems as porting the applications to every different TM interface requires significant effort.

STAMP does not provide lock based implementation for comparison purposes and is implemented with precise knowledge about the shared data and when it is concurrently accessed. This approach assumes a perfect compiler that can filter every shared variable from non-shared, which is not the likely case. STMBench7 cannot be used for HTM as it has mainly large data structures and long running transactions. SPLASH-2 is suitable to underline the performance of HTM only as it has mostly short transactions with small read and write sets that can fit in the hardware caches. But the short transactions in SPLASH-2 would incur significant overhead in STMs that cannot be amortized.

In this paper we present WormBench - a configurable (customizable) TM workload written in C#, designed from the ground up to evaluate and verify the correctness of TM systems. The design approach of WormBench is driven by the problem of how to evaluate the TM system as another tool for providing synchronization in multi-threaded applications. All the concurrency control mechanisms like locks, message passing and also transactions exists as means for efficiently solving the synchronization problems in parallel applications. Probably if we didn't have these synchronization problems such as concurrent access to a shared data, we would not need concurrency control at all. WormBench's implementation does not depend on a particular STM or HTM interface and the critical sections in the code are expressed in terms of the language-level atomic blocks. It assumes that the compiler or runtime system translates these into the appropriate concurrency control operations on a TM implementation. This way it can be used also to test the effectiveness of optimizations performed by TM-enabled compilers which are just starting to appear in the horizon [8, 6, 1, 14]- something which has been neglected so far . WormBench is highly configurable and can be configured to reproduce a certain runtime behavior that might be general enough and represent a typical multi-threaded application or specific that stresses a particular aspect of the TM system such as dealing with overflowing transactions.

The idea of WormBench is inspired by the popular Snake game. In the application several Worms, each driven by a dedicated thread moves within a shared environment - BenchWorld (abstraction of a matrix). Each move consists of several critical operations accompanied by computation. Worms can be grouped so that they recreate complex synchronization scenarios where multiple threads are involved. By changing the parameters of the applications such as the type of performed computation, the size of the BenchWorld and the Worm, one can devise a different run configuration which has different transactional and runtime characteristics. In this paper we describe the characteristics of every operation that a worm may apply and later analyze them altogether by studying 40 different run-configurations. Then we demonstrate by example how WormBench can be configured to exhibit almost the same transactional behavior that the genome application from STAMP benchmark has. Thus, WormBench is able to mimic different existing TM applications through reconfigurations.

The goal of WormBench is to help TM researchers easily create transactional workloads that they can use to verify and evaluate the efficiency of their TM systems and the compiler infrastructure that sits between the programming language and the TM. Using WormBench, one can develop a set of representative run configurations that has the transactional behavior of a typical multi-threaded application. Then use these run configurations as a baseline to compare different TM systems among each other and against the lock based version. Also, as being general enough, WormBench can be configured as a workload to stress a low level implementation detail in the TM system such as frequent read set overflows.

The rest of the paper follows with the requirements section where we discuss the issues that drive the design of WormBench. After that we describe the design and implementation of WormBench with greater details in Section 3. In Section 4, we present and analyze a set of different run configurations that show how the transactional characteristics changes based on the different parameters. In Section 5 we give an example run configuration that has very similar transactional characteristics to the genome application in STAMP. In Section 6 we discuss the related research. Conclusion follows in Section 7 and we finish by discussing the open issues in section Future Work.

2. REQUIREMENTS

Because Transactional Memory is about concurrency control, the main requirements for a representative Transactional Memory workload should include the common synchronization problems that exist in multi-threaded applications. In this way, we would be able to see how a given synchronization problem is solved by conceptually different techniques - locks which are blocking versus transactions which are non-blocking and compare them against each other. And also, to be able to compare different Transactional Memory systems, it is required that a representative workload should consider the essential features of the Transactional Memory system. This section discusses the synchronization problems and the TM relevant metrics that should be considered when building a representative workload or a suit of workloads to evaluate Transactional Memory systems.

2.1 Synchronization Problems

The necessity of having concurrency control is because of the common synchronization problems that exist in multi

threaded applications. The typical synchronization problems that can be seen in these applications and that a representative synthetic TM workload should have an instance of, are:

- Object access serializability [2] - managing a concurrent access to a shared data. This is the typical scenario when we guard the access of a shared variable by lock;
- Barrier synchronization - making group of threads to wait at certain point of execution until all (or group) of them arrive there;
- Two phase locking and its derivatives [19] - a locking protocol which attempts to provide the efficiency of fine grain locking and avoiding dead-lock by enforcing a given pattern;
- Dining philosophers [20] - is a synchronization problem that demonstrates the deadlock problem;
- Multiple granularity locking [11] - a fine grain locking technique used to lock a region in a hierarchical data structures like trees.

2.2 Metrics

To be able to compare different TM system between each other and also TM systems against lock based implementations, a representative workload application should clearly identify a set of metrics that can be used to quantitatively evaluate the performance of different TM systems. These metrics should source from the application and not be specific to a particular design or implementation style of any TM system (HTM or STM). Based on the metrics used in the existing TM research, we decided to collect the following runtime metrics in an application:

- Execution time of the application;
- Number of entered critical sections (i.e. atomic blocks);
- The ratio between reads and writes (e.g. 90% reads and 10% writes);
- Size of the accessed data structures;
- The execution time spent while in a critical section (short transactions vs. long transactions);
- Number of successfully committed transactions;
- Number of reads and writes per transaction;
- Prevalent type of operations in the application (I/O, CPU, memory); and
- Locality of memory references (spatial vs. temporal).

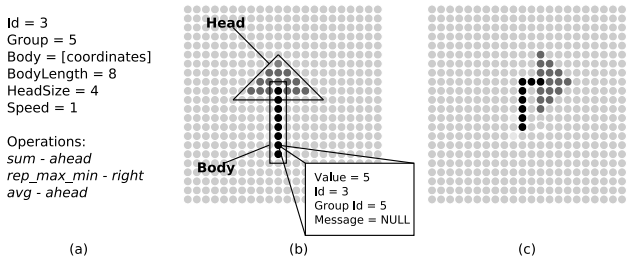

Figure 1: The main components in the WormBench application. (a) run configuration; (b) the position of the worm before performing the operations in the run configuration; (c) position of the worm after performing the operations in (a).

3. DESIGN AND IMPLEMENTATION

The idea for WormBench is inspired from the Snake game. The application has two main data structures - *BenchWorld* and *Worm*. In the application, several Worms move in the BenchWorld and execute worm operations from an user specified stream (see Figure 1). Each cell in BenchWorld is a *BenchWorldNode* struct which packs several data: (1) a value of the node, (2) the reference to the worm that is on this cell, (3) a reference to the group to which the worm on this cell belongs to, (4) and a message for the next worm that will pass from this cell. Worms are active objects meaning that every Worm object is associated with one thread. A Worm object has several attributes: id, group, speed, body, and head. *Id* is a unique identifier to distinguish the worm from the other worms, *group* is a reference to a Group object that groups several worms together. The rationale behind the notion of group is to be able to create synchronization scenarios where several worms act together to achieve a common task. The *speed* attribute is used to tell how fast the worm to advance (e.g. 1 cell per move). The *body* of the worm is the set of the cells from the BenchWorld where the worm steps on. The *head* of the worm represents a set of nodes from the BenchWorld that the worm uses as input to every worm operation, and the result of the performed computation is stored in a private buffer for verification purposes. Worms are initialized with a stream of *worm operations* (see Figure 1-a) that they should perform on every move. Every move is completed in three steps: (1) read the cells below the head, (2) perform a worm operation on the head values, (3) and move its body forward. Each of these three steps involves a critical operation and is either synchronized with an atomic block (TM system) or with a global lock (preset at compile time). Reading the values below the head of the worm involves computing the worm orientation and the head coordinates. When the head values are read, the next worm operation from the operations stream is applied to these head values and the produced result is stored in a private buffer for verification purposes. When it's time to advance forward, the worm updates the group field of every node constituting its body. In the transactional version of the benchmark this is a *conditional atomic* block which ensures that worms belonging to other groups cannot cross through each other. Every attempt of crossing would result in aborting the attacker transaction and blocking until the other worm moves its body out of the occupied node.

Currently we have implemented 15 operations[1] that Worms

[1]For space reasons we are not able to describe every operation in this paper. Instead the reader is referred to the more

Table 1: The effect of the HeadSize on read and write

Op	1		2		4		8	
	R	W	R	W	R	W	R	W
1	11	3	11	4	11	6	11	10
2	11	3	11	4	11	6	11	10
3	11	3	11	4	11	6	11	10
4	11	3	11	4	11	6	11	10
5	11	3	11	4	11	6	11	10
6	14	5	15	5	14	7	14	11
7	14	5	15	5	14	7	14	11
8	14	5	15	5	14	7	14	11
9	14	5	15	5	14	7	14	11
10	14	5	15	5	14	7	14	11
11	14	5	15	5	14	7	14	11
12	16	4	16	5	16	7	16	11
13	16	4	16	5	16	7	16	11
14	11	3	11	4	11	6	11	11
15	11	3	11	4	11	6	11	11

Table 2: The effect of the BodyLength on read and write

Op	1		2		4		8	
	R	W	R	W	R	W	R	W
1	11	3	14	3	26	3	74	13
2	11	3	14	3	26	3	74	13
3	11	3	14	3	26	3	74	13
4	11	3	14	3	26	3	74	13
5	11	3	14	3	26	3	74	13
6	14	4	17	4	29	4	77	4
7	14	4	17	4	29	4	77	4
8	14	4	17	4	29	4	77	4
9	14	4	17	5	29	5	77	5
10	14	4	17	5	29	5	77	5
11	14	4	17	5	29	5	77	5
12	16	4	19	7	31	19	79	67
13	16	4	19	7	31	19	79	67
14	11	3	14	3	26	3	74	13
15	11	3	14	3	26	3	74	13

apply in step (2). These operations have different transactional characteristics that are described in Table 1 and Table 2. Both, Table 1 and Table 2, show respectively how the change on the Worm's body length and the head size affect the transactions' read (R) and write (W) set per each Worm operation. When the head size is constant and only the body length changes, the read set remains constant and the write set increases linearly. On the other hand, when the body length is fixed to 1 and the head size changes, both the read and write sets are affected and the read set increases superlinearly. Any combination of these operations with the body length and head size of the worms could give theoretically infinite number of TM specific runtime configurations.

Table 3 summarizes the execution distribution of the Worm operations for 4 different body length and head size setups ran over 800,000 moves. The first column is the worm operation, the second column is the execution distribution when the body length and head sizes are 1-1 (B[1.1] means body length is 1, H[1.1] means the head size is 1), the third column is for worms with body length and head size of 4-4, the fourth column is when the body length and head size is 8-8 and the fifth is when the body length and head size is randomly selected in range [1, 8]. Also, the increase in the head size is reverse-proportional to the WormBench throughput (execution time). Meaning that, by increasing the head size we can obtain longer transactions suitable to test STMs and by decreasing the head size we can obtain shorter transactions suitable to test HTMs. The relationship between the head size and the throughput can be seen in Figure 3 and Figure 4 discussed in more details in Section 4.

When WormBench starts, it is initialized with a *run configuration* provided as input by the user. The *run configuration* defines: (1) the size of the BenchWorld (the size of the underlying matrix) and its initialization, (2) a common stream of worm operations; (3) the number of worms to create; (4) and for each worm: id, group id, body size and the location of the body on the BenchWorld, head size, speed, and a range from a common stream of worm operations that the worm has to perform on every move. By utilizing the summarized information in Table 1, Table 2 and Table 3, we can directly control the read set, write set and the conflict rate. Also, assigning each worm a specific stream of operations to perform, we can coarsely control the conflict rate between the transactions. For example, a stream of operations that leads all worms in a common point within the BenchWorld would result into a large number of aborts. Further, by properly using the messaging and the group notion, we can recreate instances of the synchronization problems described in Requirements section.

derailed technical report of WormBench [23].

Table 3: Execution time distribution of Worm operations

Op	B[1.1]H[1.1]	B[4.4]H[4.4]	B[8.8]H[8.8]	B[1.8]H[1.8]
1	0.42	0.43	0.19	0.31
2	0.42	0.27	0.32	0.43
3	0.84	3.65	9.35	5.14
4	0.32	0.59	0.28	0.37
5	0.42	0.59	0.33	0.54
6	1.36	0.71	0.43	0.74
7	0.74	0.74	0.53	0.70
8	2.52	4.79	11.41	6.69
9	2.10	0.59	0.63	0.93
10	2.73	5.01	11.19	7.10
11	2.52	5.26	11.39	7.12
12	1.68	6.59	11.26	7.18
13	1.15	3.25	2.37	3.37
14	1.12	1.45	1.98	1.52
15	1.06	1.32	1.85	1.49
Total	19.39	35.24	63.52	42.60

At the end, when the execution completes, we perform an automatic correctness test (i.e. sanity check for the TM system). To verify that the TM system worked properly, we compare the sum of the matrix at the end of the execution with the sum of the matrix that was at the initialization. When computing the sum of the matrix at the end of execution we also account for the modifications done by the *replace with average* operations. These modifications are stored in worms' private buffers.

WormBench is implemented in C# language by applying the concepts of object oriented programming and is compact (940 lines of code). The code is implemented with two types of synchronizations - *transactions* (atomic blocks) and *global lock*. The synchronization type can be selected at compile time. The average sizes of the shared objects is 70 bytes and have several fields which makes it favorable for TM systems that perform the versioning in object granularity (mostly STM), cache line and word granularity (mostly HTM). The primary performance evaluation metric in the BenchWorld application is the *throughput - the total number of moves per unit time*.

In behavior and synchronization, WormBench resembles the typical multi-threaded applications where independent threads perform memory reads, do computation and update a given global state. An example could be a web server with dynamic content rendering. Where the requests of the clients are served by different threads as the memory is searched for cached pages and updated on the fly depending on the provided input by the client.

To sum up, WormBench is highly configurable. By preparing different run configuration, its runtime and transactional characteristics may by easily tuned to match those of real multi-threaded applications, as will also be shown in Section 5. Also, depending on the particular run configuration,

transactions can be short and have small read and write set which will make it more favorable to HTMs. A run configuration with large transactions that have large read and write set would be more favorable to STMs. Worms with different body length and head size may be suitable for hybrid solutions that either virtualizes HTMs or accelerate STMs.

4. ANALYZING WORMBENCH

The overall behavior of the WormBench application depends on the run configuration passed as input by the user. The runtime characteristics of the application can be altered by tuning any of the following parameters:

- Size of the BenchWorld;
- Number of worms (number of threads);
- Body length of each worm;
- Head size of each worm;
- The number and type of worm operations that each worm has to perform while moving; and
- Synchronization type - atomic, lock.

Altering any of these configuration parameters, we can prepare a run configuration to reproduce runtime behavior that might be general enough and represent a typical multi-threaded application or specific that stresses a particular aspect of the TM system such as many aborting transactions.

In this section we present several run configurations with the purpose of studying the relationship between the configuration parameters and the behavior of WormBench. We also compare the obtained results in the transactional version of WormBench with the lock based version.

4.1 Experimental Settings

We performed all measurements on a Dell PE6850 workstation with 4 dual-core x64 Intel Xeon processors with 32KB IL1 and 32KB DL1 private per-core, 4MB L2 shared between the two cores on-die, 8MB L3 shared between all cores, and 32GB RAM. During our experiments hyper-threading was enabled, thus having 16 logical CPUs. The operating system is Windows Server 2003 SP2. The processor scheduling and the memory management policies were adjusted to favor foreground applications instead of background services. To compile the WormBench source code we used Bartok compiler [8]. Bartok is an optimizing compiler and managed runtime system for the Common Intermediate Language. It has compiler level and runtime support for STM. The STM runtime library does eager version management, lazy conflict detection and reads are not visible among the threads. The memory management in WormBench is performed by a two-generation copying garbage collector.

4.2 Description of the Run Configurations

In our experiments we used a single stream of 800.000 move operations. Both the operation type and the direction to move to were randomly generated with uniform distribution of the described worm operations and the three directions

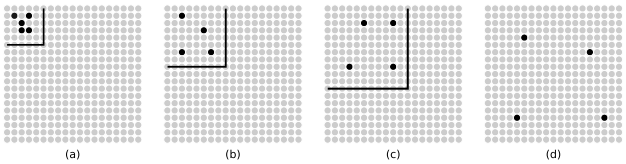

Figure 2: Using worms initialized for small BenchWorld in a large BenchWorld. (a) using worms initialized for 128x128 in 1024x2024; (b) using worms initialized for 256x256 in 1024x2024; (c) using worms initialized for 512x512 in 1024x2024; (d) using worms initialized for 1024x2024 in 1024x2024.

(ahead, left, right). To analyze the impact of the Benchworld size we used 4 different BenchWorlds with 128x128, 256x256, 512x512, and 1024x1024 sizes. To analyze how the worm's body length and head size affect the execution we used four different (body length, head size) configurations - all the worms have body length and head size 1 (indicated as B[1.1]H[1.1]), all the worms have body length and head size 4 (B[4.4]H[4.4]), all the worms have body length 8 and head size 8 (B[8.8]H[8.8]), and all the worms have both body length and head size randomly generated in range [1, 8] (B[1.8]H[1.8]). Also, we prepared four different initializations for the worms based on the underlying BenchWorld size - 128x128, 256x256, 512x512, 1024x1024. To see how the worms initialization affect the execution, we run worms initialized for smaller BenchWorld in larger BenchWorld. For example, we ran worms initialized for 128x128 BenchWorld in a 1024x1024 BenchWorld. As shown on Figure 2, worms initialized for smaller BenchWorld are relatively closer to each other and likely to be source of frequent conflicts. Based on the synchronization - we prepared two different executables with transactions and global lock. We made all the possible combinations from the so far described different configurations and ran with 1, 2, 4, 8 and 16 worms (threads). This resulted in total of 80 combinations with 400 independent runs. We repeated each of these runs 3 times and present the averaged results.

4.3 Analysis of the Run Configurations

Figure 3 shows how the throughput is affected by the different synchronization, TM (atomic) and locks. From this figure we can conclude that although the atomic version of WormBench scales, its best performance with 16 worms (threads) is less than the lock based synchronization with 1thread. The reason for this is that the STM systems incur significant overheads when doing versioning of the accessed read and write set, especially on the case when the worms body and head is 8 (B[8.8]H[8.8]) and the transaction has big read and write set. Another issue that can be observed is that the performance of lock based version degrades when ran with more than 1 thread. The reason for this is that the Bartok runtime is optimized for the case when the "lock" operation targets a lock that is not held. If the "lock" operation finds that the runtime lock has been already set by an earlier compare-and-swap operation then an OS mutex is created and thread blocked. In our case WormBench uses global lock which is most likely acquired and this way reflected negatively to the total throughput.

Figure 4 summarizes the relationship between the throughput (total number of moves per millisecond), the body length and head size, and the BenchWorld size. From the different charts (a), (b), (c) and (d) altogether is interesting to note

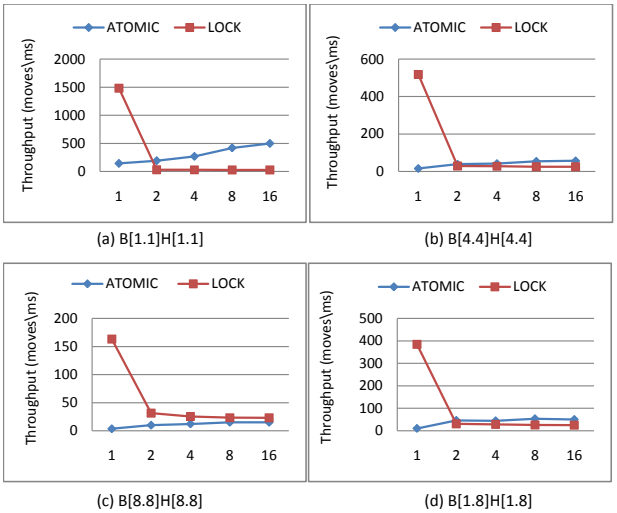

Figure 3: Comparing the performance between lock based synchronization and transactional memory synchronization (higher values are better). (a) all worms have body length and head size 1; (b) all worms have body length and head size 4; (c) all worms have body length and head size 8; (d) both the body length and head size of every worm is randomly selected from the range [1, 8].

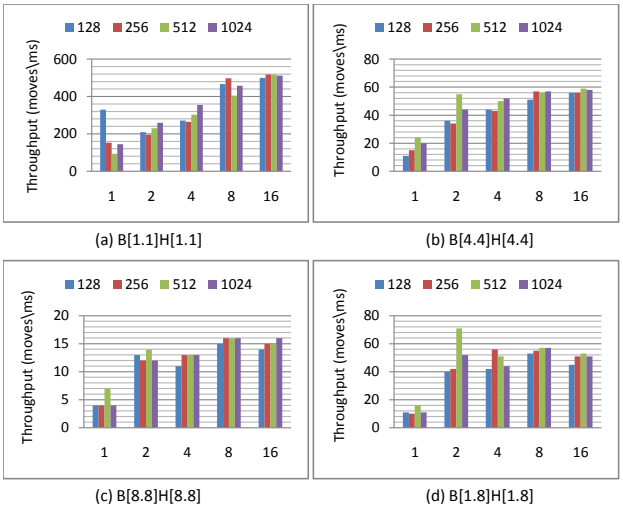

Figure 4: Relationship between throughput (total number moves per millisecond), BenchWorld size and the worm's body length and head size (higher values are better). (a) all worms have body length and head size 1; (b) all worms have body length and head size 4; (c) all worms have body length and head size 8; (d) both the body length and head size of every worm is randomly selected from the range [1, 8].

here that the increase in the body length and head size have significant impact on the throughput. The obvious reason for this is that when the body length and head size becomes larger (especially head size, which has a $O(n^2)$ impact) the input to the worm operations become larger and they spend more time doing computation. For example, in the case with a head size of 1 summing has only one node to add but in the case with head size of 8 has 64 nodes. Another reason is that when body length and head size increase, transactions become larger and their working set increases super linearly. The overhead for maintaining big read and write sets along with the increased probability for aborts becomes higher. This can be better seen in Figure 3-c with B[8.8]H[8.8], when the transactional version of WormBench always performs worse because of the overhead incurred by the versioning and frequent aborts.

Figure 5 shows the ratio between the read and write set for the different worms' sizes (body length and head size). The results are averaged across the different sizes of the BenchWorlds. In the analyzed run configurations, there are insignificant differences in the results between the different worms and mostly the reads are about 80% and writes are about 20%.

Figure 6 shows the average number of the objects opened for read or write per transaction. The *unfiltered* read set and write set (denoted as *UfR* and *UfW*) represent all the objects to which the TM system attempted to access and the filtered read and write set (denoted as FR and FW) represents the actual number of objects versioned by the TM systems. For example, it may happen that one object or memory location is once versioned and later accessed again. In this case the TM system filters it and does not allocate an entry for the second access. In Figure 6 is interesting to see that although the unfiltered read and write set increases for the different sizes of the worms, the filtered set remains constant.

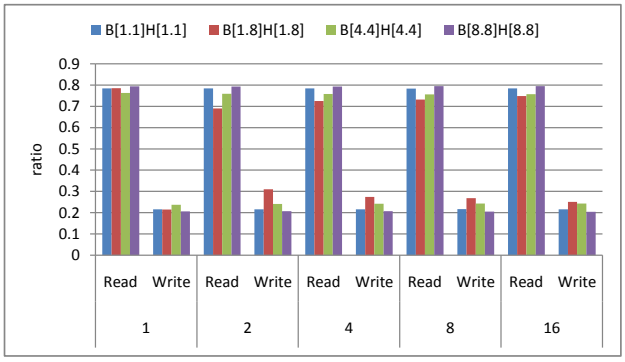

Figure 5: The ratio between the objects in the read set and writes set.

Figure 7 shows the rate of successful commits (opposite to aborts). The commit rate in all the run configuration is very high. One reason for this is mainly because of using big BenchWorlds. Based on the results in this graph, we can conclude our previous observation: since the commit rate is high, the primary factor affecting the performance of B[8.8]H[8.8] configuration is the versioning overhead.

Figure 8 shows the commit rate results of run configuration with worms initialized for BenchWorld with size 128x128 and used in BenchWorlds with larger sizes (see Figure 2). The results in this figure are different from Figure 7 since its purpose is to show how the initialization of the worms affect the commit rate. The obtained results does not significantly differ from those in Figure 7 because we initialized the worms with big worm operations streams. Consequently, this long execution has effectively decreased the impact of the conflicts occurred at the beginning of the execution when the worms were relatively closer to each other. This configuration can model a TM-execution which has phases: in the first phase it starts with a high conflict rate and con-

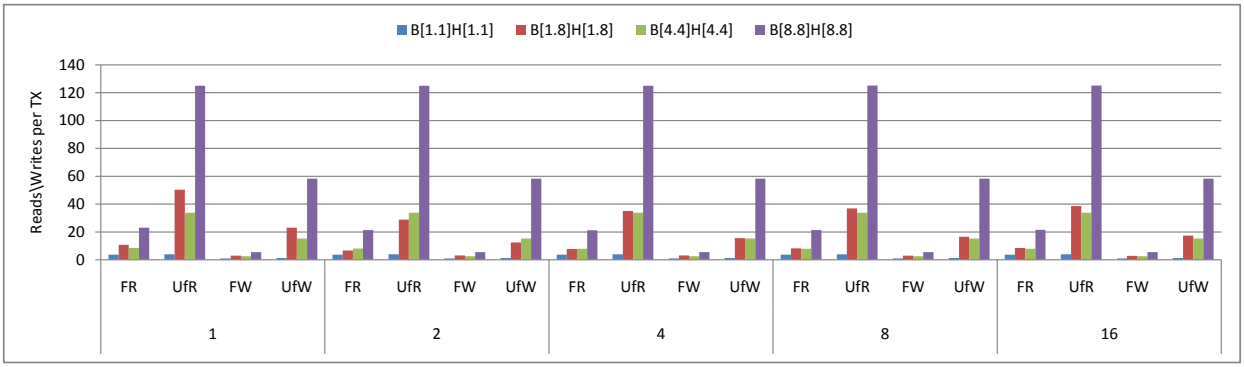

Figure 6: The number of unfiltered reads (UfR) and writes (UfW) per transaction and the number of filtered reads (FR) and writes (FW) per transaction.

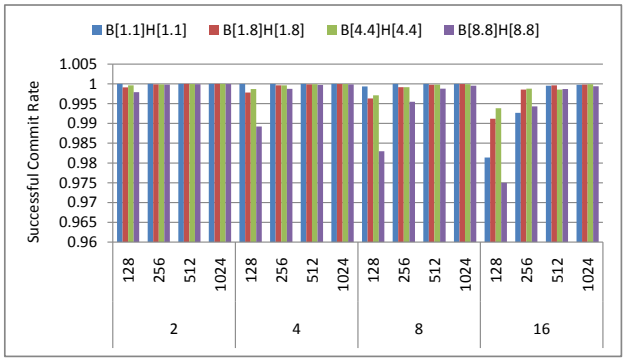

Figure 7: The average commit rate for all configurations. We omit the case for 1 worm (thread) because it is always 1.

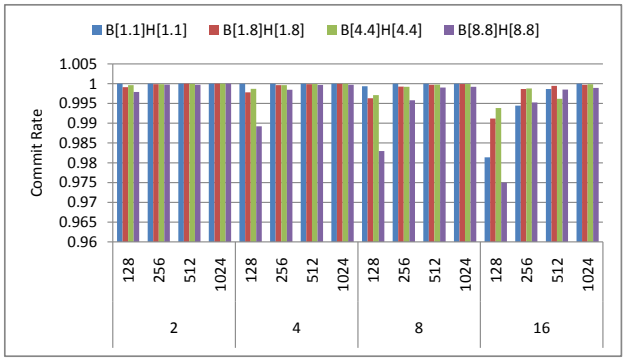

Figure 8: The commit rate when worms are initialized for BenchWorld with size 128x128 and then used in larger BenchWorlds - 128x128, 256x256, 512x512 and 1024x1024.

tinues with a lower conflict rate in the second phase. This characteristic of WormBench could be very useful in testing how well adaptive TM systems perform in the presence of changes in runtime TM-application behavior.

Based on the analyzed results in this section and the described characteristics of WormBench in the previous section, we will next show by example run configuration that WormBench can mimic the behavior of genome application from STAMP.

5. MODELING A TM APPLICATION

To demonstrate that WormBench is highly configurable we prepared a run configuration that has the similar transactional characteristics of the genome application from the STAMP benchmark. Table 4 compares the TM and runtime characteristics of the *genome* (Gen.) application and the run configuration for WormBench (WB) that mimics genome. *Read per TX* is the reads and *Write per TX* is the Writes. The commit rate and the number of reads (R) is very similar to the original values in genome. The proposed run configuration scales up closely following the speedup rate of the original application. The number of writes (W) per transaction in WormBench is a little bit higher than in the original application but a careful tuning would be possible to lower writes and at the same time keep the other parameters unchanged.

Table 4: Modeling Genome application with WormBench.

T#	Commit Rate		Read per TX		Write per TX		Speedup	
	Gen.	WB	Gen.	WB	Gen.	WB	Gen.	WB
1	1	1	36.36	31.48	1.37	1.96	1	1
2	0.998	0.998	34.26	31.60	1.37	1.96	2.18	1.4
4	0.994	0.995	37.97	31.81	1.37	1.96	3.47	2.2
8	0.985	0.987	46.219	32.30	1.37	1.96	5.43	2.87

To obtain the results shown on Table 4 we used the following run configuration:

- Worms body length = 1
- Worms head size = 4
- BenchWorld of size 52x52
- Randomly generated stream of worm operations, where the ration between the worm operations was- Operations(1:2:3:4:5:6:7:8:9:10:11:12:13:14:15) = Ration(1:1:1:0:0:2:1:1:1:1:1:1:2:0:0)

This is just a small example that demonstrates the high configurability of WormBench and how it can be used to reproduce the runtime and TM characteristics of a specific multi threaded application.

6. CONCLUSION

This paper presented WormBench - a configurable workload for evaluating Transactional Memory. It is designed and implemented from the ground up by having in mind transactional memory as a mechanism for concurrency control. WormBench is parameterized and highly configurable. By preparing a specific run configuration, one can easily obtain a runtime and TM specific behavior that closely mimics the behavior of a real multi-threaded application. We described the transactional characteristics of each critical section existing in WormBench and also analyzed their characteristics

altogether in 40 different run configurations. We demonstrated its flexibility by preparing a specific run configuration that has similar transactional behavior like the genome application from the STAMP benchmark suit.

Overall, being configurable, the goal of WormBench is to serve as a handy tool for instrumenting a specific transactional application that will stress a particular design or implementation aspect of the low level implementation detail or prepare a general enough set of run configurations that can evaluate all aspects of transactional memory from the hardware micro-architecture to the language extensions.

7. FUTURE WORK

We plan to soon release the WormbBench source code and the auxiliary tool set to the research community. As a further research work, we plan to analyze the effect of the messaging between the worms. Messaging and collaboration between worms would result in interesting synchronization problems. We, also plan to provide different type of implementations for the BenchWorld such as linked list, and sparse matrix and study their impact on the runtime and transactional behavior.

One another very interesting bullet in our future work list is implementing a tool that can automatically build a run configuration provided with the specific transactional characteristics.

8. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their useful comments. This work is supported by the cooperation agreement between the Barcelona Supercomputing Center - National Supercomputer Facility and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625, by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC) and by the European Commission FP7 project VELOX (216852). Ferad Zyulkyarov is also supported by a scholarship from the Government of Catalonia.

9. REFERENCES

[1] W. Baek, C. Cao Minh, M. Trautmann, C. Kozyrakis, and K. Olukotun. The opentm transactional application programming interface. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pages 376–387. Sep 2007.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, Reading, Massachusetts, 1987.

[3] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *In 13th International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2007.

[4] W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. V. Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded page-based transactional memory. In *ASPLOSXII-2006*, pages 347–358, 2006.

[5] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *20th International Symposium on Distributed Computing (DISC)*, pages 194–208, 2006.

[6] P. Felber, C. Fetzer, U. Müller, T. Riegel, M. SüSSkraut, and H. Sturzrehm. Transactifying applications using an open compiler framework. In *ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2007.

[7] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: A benchmark for software transactional memory. In *Second European Systems Conference EuroSys 2007*, 2007.

[8] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *Programming Language Design and Implementation (PLDI)*, June 2006.

[9] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20th Annual International Symposium on Computer Architecture (ISCA)*, pages 289–300, May 1993.

[10] M. Herlihy, L. Victor, M. Moir, and N. S. I. Scherer. Software transactional memory for dynamic-sized data structures. In *22nd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, July 2003.

[11] S.-Y. Lee and R.-L. Liou. A multi-granularity locking model for concurrency control in object-oriented database systems. In *Transactions on Knowledge and Data Engineering*, pages 144–156, Feb 1996.

[12] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *2008 IEEE International Symposium on Workload Characterization*, September 2008.

[13] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, and K. Kozyrakis, Christos ans Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *34th Annual International Symposium on Computer Architecture (ISCA)*, June 2007.

[14] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cowine, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for c/c++. In *OOPSLA*, October 2008.

[15] C. Perfumo, N. Sonmez, S. Stipic, A. Cristal, O. Unsal, T. Harris, and M. Valero. The limits of software transactional memory (stm): Dissecting haskell stm applications on a many-core environment. In *Computing Frontiers 2008*, May 2008.

[16] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *32nd Annual International Symposium on Computer Architecture (ISCA)*, June 2005.

[17] T. Riegel, P. Felber, and C. Fetzer. Dynamic performance tuning of word-based software transactional memory. In *13th PPoPP*, February 2008.

[18] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *39th International Symposium in Microarchitecture*, December 2006.

[19] A. S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1994.

[20] A. S. Tanenbaum. *Modern Operating Systems (2nd Edition)*. Prentice Hall, 2001.

[21] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *ISCA-22*, pages 24–38, June 1995.

[22] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. M. S. Mark D. Hill, and D. A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *HPCA-13*, Feb 2007.

[23] F. Zyulkyarov, S. Cvijic, O. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. Wormbench - technical report. Technical Report UPC-DAC-RR-CAP-2008-23, Universitat Politechinca de Catalunya, https://gw.ac.upc.es/go/gsi/reports/2008/46/WormBench-techrep.pdf, August 2008.