The Limits of Software Transactional Memory (STM): Dissecting Haskell STM Applications on a Many-Core Environment

Cristian Perfumo
Barcelona Supercomputing Center / Universitat Politècnica de Catalunya
Barcelona, Spain
cristian.perfumo@bsc.es

Nehir Sönmez
Barcelona Supercomputing Center / Universitat Politècnica de Catalunya
Barcelona, Spain
nehir.sonmez@bsc.es

Srdjan Stipic
Barcelona Supercomputing Center / Universitat Politècnica de Catalunya
Barcelona, Spain
srdjan.stipic@bsc.es

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

Adrián Cristal
Barcelona Supercomputing Center
Barcelona, Spain
adrian.cristal@bsc.es

Tim Harris
Microsoft Research
Cambridge, UK
tharris@microsoft.com

Mateo Valero
Barcelona Supercomputing Center / Universitat Politècnica de Catalunya
Barcelona, Spain
mateo@ac.upc.edu

ABSTRACT

In this paper, we present a Haskell Transactional Memory benchmark to provide a comprehensive application suite for the use of Software Transactional Memory (STM) researchers. We develop a framework to profile the execution of the benchmark applications and to collect detailed runtime data on their transactional behavior, running them on a 128-core multiprocessor. Using a composite of the collected raw data, we propose new transactional performance metrics. We analyze key statistics related to scalability, atomic sections, transactional events, overall transactional overhead and the relative hardware performance, accordingly drawing conclusions on the results. Our findings advance our comprehension on the STM runtime and the characteristics of different applications under the transactional management of the pure, functional programming language, Haskell.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement techniques

General Terms

Measurement

Keywords

Transactional Memory, Haskell, Instrumentation

1. INTRODUCTION

Chip Multiprocessors are already a standard in the market and they demand efficient use of parallelism and easier methods in programming shared-memory parallel architectures. In this scenario, while traditional mechanisms such as lock-based thread synchronization tricky to use and non-composable, researchers are investigating alternative programming models. One serious candidate are language constructs built over transactional memory in which a set of memory accesses are performed as-if atomically to the heap.

Research on Transactional Memory (TM) has been conducted for the last fifteen years with different flavors, semantics and implementations [13, 23, 8]. While some basic intuitive conclusions can be reached after observing execution times and speedup analysis of TM applications, it would be much more useful to look deeper inside such applications to extensively inspect the most relevant characteristics of TM, such as the number of committed transactions, the rate of aborts and read-write set sizes. Although it is not absolutely necessary, when a programmer runs her application written under a transactional paradigm, she might like to know some internal details of the execution. What is absolutely mandatory is that a researcher knows very well how the engine works, because she is trying to make observations on the results to eventually make something faster, better or easier.

Typical implementations of TM work by executing transactions optimistically in the hope that different threads' transactions will access different sets of memory locations. If that assumption is correct then the transactions commit their updates. Otherwise the transactions are rolled back. Rajwar and Larus [16] provide a summary of the wide range of implementation techniques being explored; for example, whether conflicts are prevented from happening (or detected after the fact), and whether tentative updates are made inplace as transactions run, or only made when a transaction successfully commits.

As hinted above, in the particular case of TM, some questions arise when an application is executed: Does my application suffer from a high rollback rate? Is it spending a lot of time in the commit phase? What is the overhead that the transactional runtime introduces? How big are the readset and the writeset? Is there any relationship between the number of reads and the readset? What about writes? What is the transactional read-to-write ratio? What would be the trend like with more processors? One major aim of this work is to profile a set of transactional Haskell applications and to draw conclusions out of the metrics obtained. In order to accomplish this, the Haskell Runtime System (RTS) has been modified by inserting monitoring probes that extract application runtime data relevant to the software transactional execution. Additionally, the Performance Application Programming Interface (PAPI) library [5] is used in an attempt to further dissect transactional behaviour using hardware performance counter data.

Another motivation for this work is the dearth of Transactional Memory benchmarks. Although several benchmarks have been developed for future multi- and many-core Chip Multiprocessors (CMP) [17, 3, 19], none of the applications in those benchmarks use TM. Pre-TM era Haskell benchmarks exist [21], and recently a one-application highly-tunable TM benchmark was developed for imperative languages: STMBench7 [11], alongside the STAMP [6] benchmarks. No TM benchmark exists for Haskell; an appropriate environment to test new TM ideas given its small and easy-to-modify runtime system.

Two recent studies have also attempted to profile transactional executions. Chung et al.'s work [7] contains a diverse set of parallel applications to analyze transaction lengths, readset-writeset sizes, transaction nesting and I/O. They do not evaluate the frequency of transaction rollbacks since they do not perform the experiments on a specific STM, instead they introduce transaction boundaries to monitor the critical sections of the the lock based models with the premise that the parallelism and the synchronization patterns are unlikely to change on a switch to transactional models. Bobba et al. [4], classify HTM in terms of conflict detection, version management, and conflict resolution, and proceed to categorize problematic cases. In an HTM simulation environment, they observe readset-writeset sizes, dissecting executions as aborting, aborted, committing, stall, backoff, barrier, transactional, and non-transactional. Their simulations do not look at number of transactional writes and reads. In terms of hardware performance counters on functional programs, the work on [20] makes a good attempt to look at the cache performance of some of the lazy functional programs of the nofib benchmark [21] on the Glasgow Haskell Compiler (GHC), however the work does not address TM in any way.

In comparison with the above work, this paper analyzes programs developed for TM from the ground up, instead of being converted from lock-based versions. To the best of our knowledge, this is the first work in which STM applications are run and analyzed on multiprocessors with large core count. Moreover, no previous attempt to profile Haskell STM applications has been carried out.

In particular, the contributions of this paper are as follows:

- A Haskell STM application suite that can be used as a benchmark by the research community is presented.
- The Haskell runtime system is instrumented with the PAPI library and manually-inserted counters to collect information on the transactional portions of each program, read/writesets, commit and abort rates, as well as the runtime overheads on the applications. To identify the limits of Software Transactional Memory, the transactional behavior was observed and analyzed on up to 120-core executions on a ccNUMA system. The hardware performance of Haskell STM was observed to understand the bottlenecks of STM. Specifically, the cache performance and stalls were inspected to make useful claims about scalability issues.
- Based on the collected raw transactional information, new metrics such as commit phase overhead, read-to-write ratio, wasted time, useful work and cache accesses per transactional access are derived, which could be used to characterize STM applications. This information could be used by researchers in future work to test the effectiveness of their STM ideas.

The rest of this paper is organized as follows: Section 2 gives some background on TM in Haskell, section 3 describes the applications in the benchmark suite, section 4 discusses the statistics obtained from the instrumentation of the runtime system, section 5 is about scalability issues and finally section 6 concludes and looks into future work.

2. BACKGROUND: TRANSACTIONAL MEMORY IN HASKELL

The Glasgow Haskell Compiler 6.6.1 (GHC) [1] provides a compilation and runtime system for Haskell 98 [9], a pure, lazy, functional programming language. GHC natively contains STM functions built into the Concurrent Haskell library [14], providing abstractions for communicating between explicitly-forked threads. As Harris et al. state in their work [12], STM can be expressed elegantly in a declarative language, and moreover, Haskell's type system (particularly the monadic mechanism) allows threads to access shared variables only inside a transaction. This useful restriction is more likely to be violated under other programming paradigms, for example, as a result of access to memory locations through the use of pointers.

Although the core of the language is very different to other languages like C# or C++, the actual STM operations are used in a simple imperative style and the STM implementation uses the same techniques used in mainstream languages [12]. Haskell STM has the attractions that (i) the runtime system is small, making it easy to make experimental modifications, (ii) Haskell is one of the first languages that integrates STM in its mainstream distribution. The STM support has been present for some time, leading to a number of example applications using transactions; indeed, leading to applications which have been written by "ordinary" programmers rather than by those who built the Haskell STM implementation.

Threads in Haskell STM communicate by reading and writing transactional variables [12], or `TVar`s, using a set of transactional operations, including allocating, reading and writing `TVar`s, namely the functions `newTVar`, `readTVar` and `writeTVar`, respectively, as it can be seen on Table 1.

Running STM Operations	TVar Operations
atomically::STM a->IO a	newTVar::a->STM(TVar a)
retry::STM a	readTVar::TVar a->STM a
orElse::STM a->STM a->STM a	writeTVar::TVar a->a->STM()

Table 1: Haskell STM Operations

STM provides a safe way of accessing shared variables among concurrently running threads through the use of monads [13], allowing only I/O actions in the `IO` monad and STM actions in the `STM` monad. Programming using distinct STM and I/O actions ensures that only STM actions and pure computation can be performed within a memory transaction (which makes it possible to re-execute transactions transparently), whereas only I/O actions and pure computations, and not STM actions, can be performed outside a transaction. This guarantees that `TVar`s cannot be modified without the protection of `atomically`, and thus separates the computations that have side-effects from the ones that are effect-free. Utilizing a purely-declarative language for TM also provides explicit read/writes from/to mutable cells; furthermore, memory operations that are also performed by functional computations are never tracked by STM unnecessarily, since they never need to be rolled back [12].

Transactions in Haskell are started by means of the `atomically` construct. Since Haskell currently implements lazy conflict detection, when a transaction is finished it is validated by the runtime system that it was executed on a consistent system state, and that no other finished transaction may have modified relevant parts of this state in the meantime [12]. In this case, the modifications of the transaction are committed, otherwise, they are discarded. The Haskell STM runtime maintains a list of accessed transactional variables for each transaction, where all the variables in this list which were written are called the "writeset" and all that were read are called the "readset" of the transaction. It is worth noticing that these two sets can (and usually do) overlap.

Operationally, `atomically` takes the tentative updates and applies them to the `TVar`s involved, making these effects visible to other transactions. When `atomically` is invoked, this method deals with maintaining a per-thread transaction log that records the tentative accesses made to `TVar`s during the "work phase" where the actual computational work takes place. Later in the "commit phase", a global lock[1] is acquired and the validation (first part of the two-phase commit) is performed: going through the readset checking each variable with its local original value that was obtained at the start of the transaction. If all `TVar`s that are read are consistent, all new values are actually committed into the memory (the second part of the two-phase commit) and then the lock is released.

In case a concurrent transaction has committed conflicting updates, the writeback cannot be performed. Instead, the rollback mechanism takes place discarding the uncommitted intermediate values and restarting the execution.

For example, Figure 1 shows a simple example where `atomically` is used in order to increment a counter inside a transaction. The counter is a `TVar` that is accessed with the operations `readTVar` and `writeTVar`, to perform a transactional read and a transactional write.

```
atomically (do
   { theValue <- readTVar tValue
   ; writeTVar tValue (theValue + 1) })
```

Figure 1: Shared variable update example

3. ANALYZED APPLICATIONS

The transactional analysis contains 10 applications[2], some of which were executed with different parameters to provide with different execution patterns, totaling 18 experiments. Some of these applications have been written by people who are not necessarily experts on implementation details of STM, whereas others have been developed by the authors of this work who have worked using the Haskell STM and modifying the GHC runtime for some time. Both kinds of applications are interesting because the former can show the way that regular programmers (the ultimate target of the whole TM research) use TM and on the other hand, a programmer with a deep knowledge on TM can explore corner cases, force specific situations or simply augment the diversity by adding applications that are intended to have a variety of features which are different from the common ones.

In the set of applications in Table 2, all the runs are arranged according to the execution environment to fit exactly one thread per core so that when, for example, four cores are used, each application spawns exactly four threads.

The applications marked with (CCHR) were taken from a Concurrent Constraint Handling Rules implementation [15], where the basic idea is to derive a set of rules and apply them until the most simplified expression possible is obtained. In order to reach the goal, each of these applications stores the current list of rules as shared data and accesses them in a transactional way. For example, the Unionfind algorithm is used to maintain disjoint sets under union efficiently. Sets are represented by trees, where the nodes are the elements and the roots are the representative of the sets [10]. Even if the results turn out to be different among CCHR programs, they all share the same principle of execution: they define a set of rules and based on the inputs they create the initial rules that are derived until getting the most simplified one, which is the output of the application. The active rules are maintained in an execution stack. Active rules search for possible matching rules in order and they become inactive when all matching rules have been executed or when the constraint is deleted from the store. In order to allow thread-level parallelism, several execution stacks are simultaneously maintained, meaning that multiple active constraints can be evaluated at the same time. The CCHR applications also balance the workload by moving constraints from one execution stack to another.

SingleInt is a simple program that updates an integer variable that is shared among the threads. It tries to perform in parallel a task that is inherently sequential. Since it updates the shared structure (integer) and almost no further computation is carried out, its performance degrades as the number of cores increases due to extremely high contention. This observation is analyzed in depth in the next section.

Linked list programs atomically insert and atomically delete an element in a shared list, whereas the binary tree (BT) programs do so on an unbalanced tree. The work-

[1] GHC also allows another way of performing the commit phase in a finer-grained fashion: by acquiring a lock per TVar. This approach has a different set of advantages and drawbacks and is not in the scope of this paper.

[2] Even though Haskell is widely used, its community seldom developes large-footprint applications, unlike other programming languages.

Application	Description	# lines	# atomic
BlockWorld	Simulates two autonomous agents, each moving 100 blocks between non-overlapping locations (CCHR).	1150	13
GCD	A greatest common divisor calculator (CCHR).	1056	13
Prime	Finds the first 4000 prime numbers (CCHR).	1071	13
Sudoku	A solver of this famous game (CCHR).	1253	13
UnionFind	An algorithm used to efficiently maintain disjoint sets under union, where sets are represented by trees and the nodes are the elements (CCHR).	1157	13
SingleInt*	A corner-case program that consists of n threads incrementing a shared integer variable for a total of 200,000 times. Therefore, the accesses to the critical section are serialized because concurrent updates to the same variable is not possible.	82	1
LL*	A family of singly-linked list applications inserting and deleting random numbers. There are three list lengths: 10, 100 and 1000.	250	7
LLUnr*	Same as LL (above), but using `unreadTVar` [25], a performance improvement that uses early release [24]. It lets a transaction forget elements that it has read, causing a smaller readset, faster commits, but also possible race conditions if it is not used properly.	250	7
BT*	A family of binary trees applications inserting and deleting random numbers. There are three maximum capacities of the trees: 100, 1000 and 10000 elements.	262	7
BTUnr*	Same as BT (above), but using `unreadTVar`.	265	7

Table 2: Description of the applications used in the benchmark, number of lines of code and atomic blocks (the ones marked with an asterisk were written by the authors of this work)

load is equally divided among the threads and always totals 16,000 operations. Lists of different lengths (10, 100, 1000) and trees with different capacities (100, 1000 and 10000 elements) were analyzed given that the behavior is intuitively expected to be different. Two flavors of each data structure exist in the benchmark: the regular, and the `unreadTVar`-optimized [25]. Since the next node links (right and left nodes in the case of trees) are implemented as `TVar`s, the regular version collects are many elements in its readset as the elements traversed, whereas the `unreadTVar`-optimized version always maintains a fixed, small-sized readset.

3.1 Initial runtime analysis

Before starting to examine detailed internal transactional behavior of the applications, it is important to have an idea about their execution times. Figure 2 plots runtimes normalized to the single-core version for 1, 2, 4, 8, 16, 32, 64 and 120 cores (in the 128-core machine, 8 were reserved for system administration), making comparisons among the scalability of different programs straightforwardly visible.

Of the applications that have obvious scalability problems, SingleInt lacks parallelism and naturally experiences high contention because the task it performs does not match the optimistic approach of most TM systems (including Haskell's). This application was explicitly written to conflict, so the more the conflicts, the more the rollbacks, the more the re-executed tasks, the more the cache "ping-ponging" and the more the execution time. Other two applications that suffer from performance degradation are both versions of the 10-element linked list (LL and LLUnr), especially with a large number of cores. The explanation for this phenomenon is that the number of cores exceeds the number of elements in the list, making the case of a conflict almost certain.

Most of the applications have smaller runtimes as the number of threads goes up for smaller processor counts, whereas for larger number of processors, the performance degrades. Figure 3 summarizes this, showing the number of applications that reach the maximum speedup with a given number of cores. It also shows that most applications do not scale well beyond eight cores. The possible causes of this situation are to be discussed in detail on Section 5, but first Section 4 will attempt to help understand the applications' behavior by proposing and analyzing a set of new metrics.

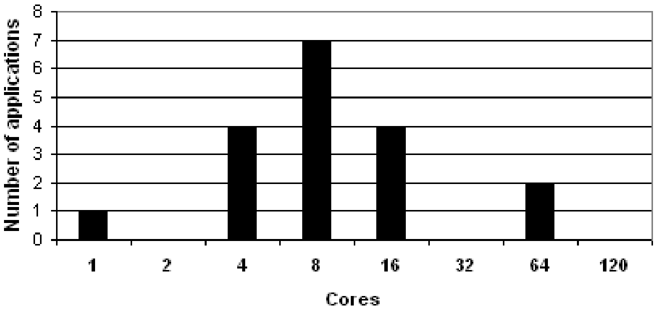

Figure 3: Histogram of the number of applications that reach the maximum speedup with a given number of cores.

No benefit is observed for large number of cores in most of the applications. There are two applications that achieve the fastest execution with 64 cores, performing twice as fast as with 8 cores: BT10000 and BTUnr10000. Note that these two applications have more inherent parallelism due to the large data structures that can be updated in parallel with low probability of conflicts. However, even for those applications, the most relevant part for analysis of obtained data ranges from 1 to 32 cores. Therefore, from this point onwards, we report detailed numbers for up to 32 cores to facilitate the readability of the paper as well as to concentrate on the more revealing data.

Given Amdahl's Law, if researchers want to propose improvements for STM management, they have to be sure that a substantial amount of the applications' runtime is spent inside a transaction: Figure 4 shows the percentage of time that each application is inside a transaction for n-core executions. Several applications spend a substantial amount of time in executing transactions, therefore the improvements made in the transactional runtime system would non-marginally improve the overall execution time.

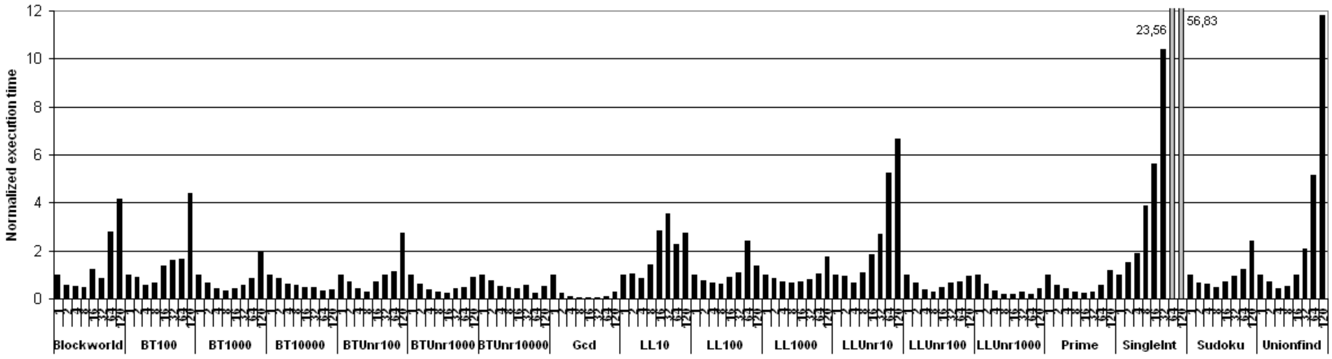

Figure 2: Execution time normalized to the single-core time

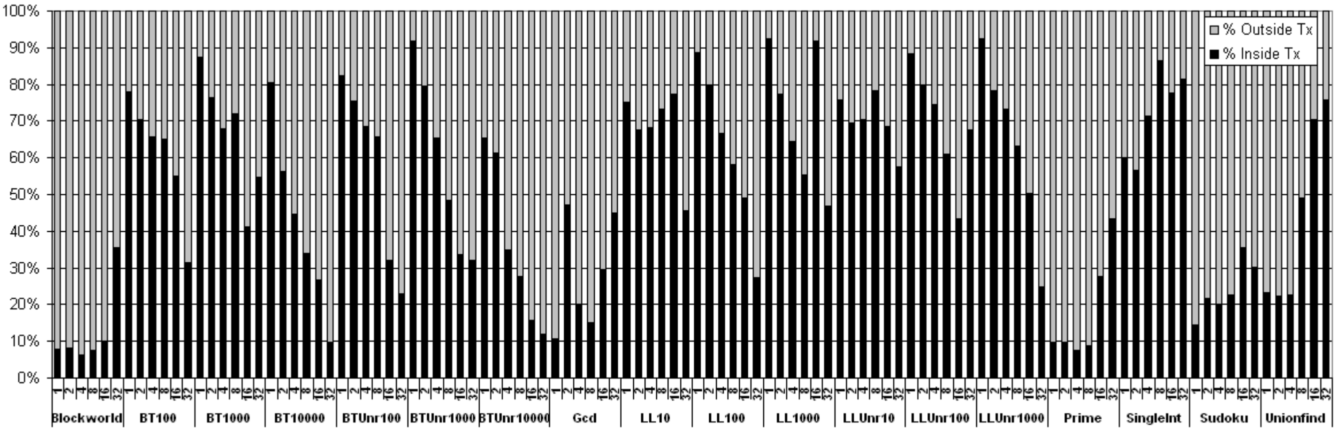

Figure 4: Percentage of the time the application is inside and outside a transaction

4. OBSERVED STATISTICS

All of the reported results are based on the average of five executions and all the experiments were run on an SGI Altix 4700 machine with 64 Dual-Core CPU, Montecito (IA-64) 1,6 GHz (128 cores total); 2 processors (4 cores) per blade, in-order; Caches: 16K L1D, 16K L1I, 256K L2D, 1M L2I, 8M L3 (all private); 512 GB RAM (Distributed shared memory, 16 GB per blade); NUMALink interconection [2].

Moreover, since Glasgow Haskell Compiler (GHC) did not support parallelism for the Itanium architecture, this feature was implemented in order to have the benchmark run on the target machine.

The statistics accumulated by the proposed extension to the runtime system are encapsulated in a function called `readTStats`, which by being invoked retrieves the list of raw statistics that are used to report the results in this work. This function resides in the file that contains all STM functions (`STM.c`) inside the GHC 6.6.1, where the values are gathered at different points in the transactional execution, i.e. while starting the transaction, while performing a read or write, or while committing. For some values, the data must be temporarily kept in the Transactional Record (`TRec`), so that new fields were added to the `TRec` structure. Although this procedure adds some overhead on the transactions, it is the unique way of achieving our objective in this context. The information about timing, cache and other events is obtained by querying hardware performance counters using the PAPI 3.5.0 library to instrument the runtime system. The update of collected statistics is done at the commit phase and the data relative to transactions that were actually committed or rolled back are accumulated depending on the result of the spaculative execution.

Out of these values we derive meaningful statistics such as reads per writes, average commit time, average work time, commit phase overhead and "wasted work", which is the ratio of the aborted work to the total work. These derived statistics could be used as metrics to gauge the transactional performance of applications using STM. Note that these metrics could also be useful for comparing the performance of various STM implementations.

In our experiments, the total STM execution time (the time spent inside an atomic block) is divided into two: the time spent for doing tha work contained inside the atomic block and the commit phase, which signifies the time taken for validating and committing or discarding the tentative changes. The start transaction phase, which is always constant and negligible independently of the characteristics of the transaction, has been omitted from our calculations.

Since Transactional Memory is a scheme that imposes the committing or rolling back of transaction sequences, the first issue while trying to monitor the transactional management is naturally the rate of rollbacks. Later in this work, other values such as the commit phase overhead, read and write sets and the percentage of wasted work will be observed. Furthermore, hardware performance counter results will be used to support intuitions where necessary.

4.1 Abort Rate

To start off, it is important to state that throughout this work, the terms abort and rollback are used interchangeably. However, the reader should be aware that in other contexts

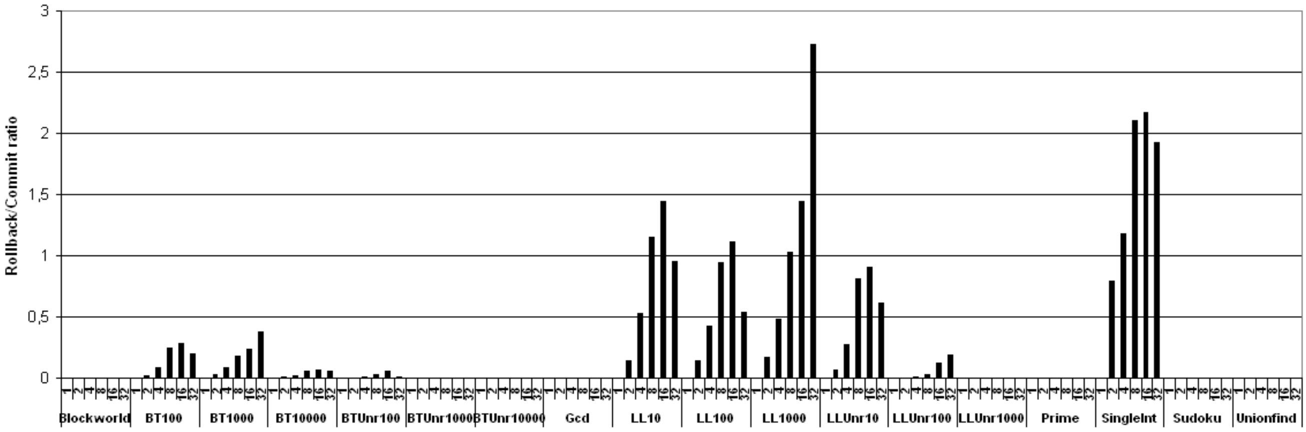

Figure 5: Rollback rate: Number of rollbacks per committed transaction

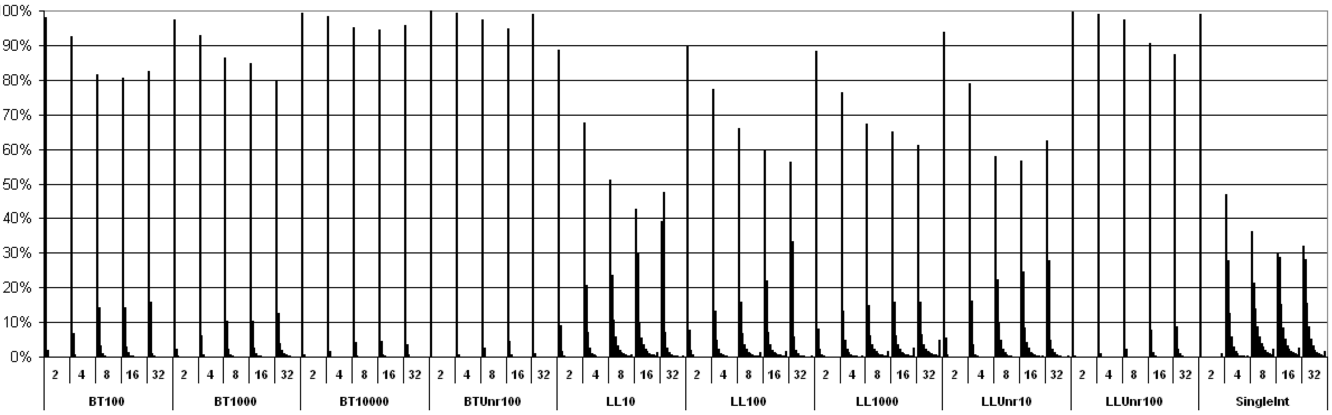

Figure 6: Rollback histograms

they might be used to mean distinct concepts.

In our set of applications, none of the programs roll back on a single core environment because of the condition previously explained: one and only one thread per core. After analyzing the abort rate (shown in Figure 5) in a multicore environment, a classification can be made based on the observed values:

- High abort rate group: The applications in this group have a rollback/commit ratio of more than 50% (in many cases, even exceeding 100%). SingleInt is a program that continuously updates its shared data, therefore aborts most of the time running on multicores, suffering from high cache thrashing. LL10, LL100, LL1000 and LLunr10 abort frequently as well. For the particular case of a ten element linked list, the high rate of rollbacks should be expected even when `unreadTVar` is used, due to the small list size being exceeded by the number of present cores. Even for larger lists, the significantly larger number of aborts is a general case for linked lists since the STM tries to look after more than what is needed by collecting all traversed elements within the readset [25].
- Medium abort rate group: The programs LLUnr100, BT100, BT1000, BT10000 and BTUnr100 have a substantial abort rate, but not as much as the previous group.
- Low abort rate group: GCD, Blockworld, Sudoku, Prime, LLUnr1000, BTUnr1000/10000 and UnionFind almost never abort (even if they do, they do it less than 1% of the time). In general, CCHR applications do not rollback very often because they utilize TM in order to take advantage of the transacted channels (which do not share much data), therefore aborting seldom. This is possible due to the fact that constraint handling rules naturally support concurrent programming [15]. The low rollback rate of BTUnr1000, BTUnr-10000 and LLUnr1000 is a consequence of the efficiency of `unreadTVar`.

In our benchmark, the number of aborts usually increases when more cores are used. Having more aborts when the number of cores (and then threads) goes up is due to the situation of having more concurrency in the application. This increases the probability of conflict and therefore, rollback.

It would also be interesting to look at how the aborts are distributed. For this purpose, the histograms in Figure 6 show the distribution of rollbacks per transaction from 0 to 10+ times for those programs that are not on the low abort rate group. It can be clearly seen that the transactions of the programs tend to be able to commit mostly on the first attempt, except for the cases of SingleInt and the LL10. It should also be apparent to see how late some transactions finally end up committing, from the last bar of each combination of application and number of cores (e.g. LL1000). Our findings show that for some applications certain transactions abort repeatedly sometimes even more than 10 times, which shows the need for further research into avoiding these “late commit” situations.

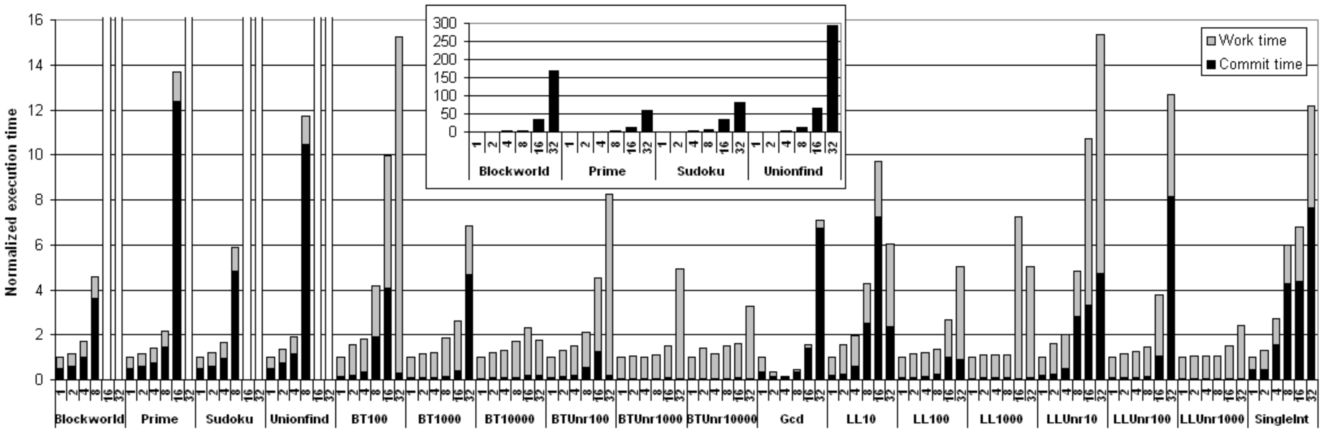

Figure 7: Commit phase overhead for committed transactions, normalized to single core time (The white bars are re-plotted in the small graph due to scale contraints)

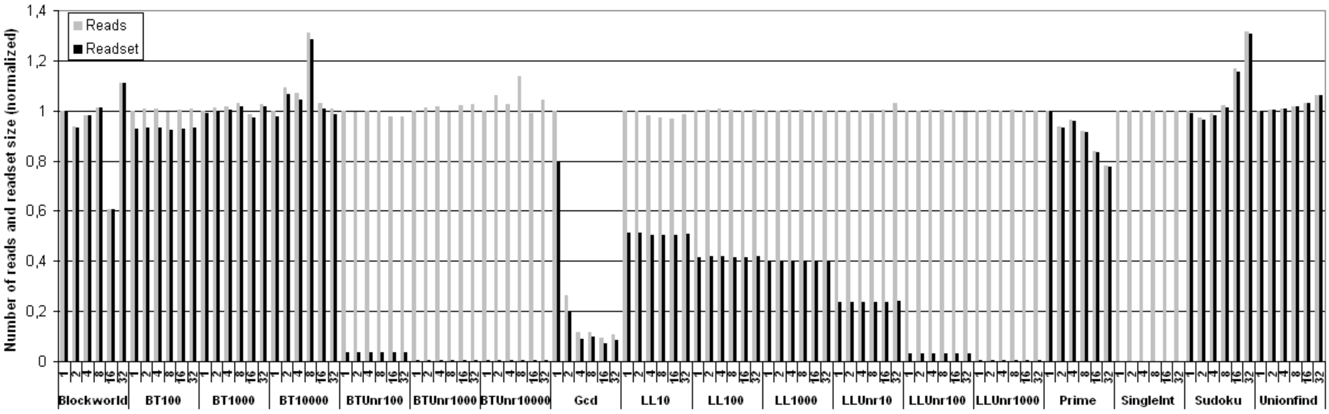

Figure 8: Number of reads and size of the readset on committed transactions

4.2 Work and commit times

Having had a general idea on the set of programs, one can now proceed to examine another important measure of transactional behavior, the commit time and its overhead.

Figure 7 illustrates the way that the time is spent when the application is within a transaction, taking into account the results for the transactions that end up committing in the end. The values are normalized to the single core execution for each application, that is, how much of the time is used to execute the transaction (work time) and how much time to commit it (commit time). We believe that the commit phase overhead, which is obtained by dividing the commit phase time by the total time, could be one of the most appropriate metrics to measure STM implementation performance, since that is the part of the transaction when the processors perform most of the communication (validating and writing data). The highest commit phase overhead per committed transaction is present in the programs belonging to the CCHR set: reaching almost 98% (visible as full-black bars in the small graph) for example in the case of UnionFind running on thirty-two cores. Specifically to provide scalability, these applications have a very large number of small transactions which avoid costly rollbacks produced by large transactions that would impact the overall performance negatively. However, this ends up producing a very large commit phase overhead. It is worth it to notice that even if eager conflict detection were being used, the costs of synchronization would be nevertheless present but they would be interleaved with transactional accesses in this case (on every read or write, a validation process would have to be carried out).

In the cases of LL and BT, until 16 cores, larger size implies less commit phase overhead. Although there is a larger readset and writeset and a greater number of reads in these programs, the commit time is observed not to rise as rapidly as the work time, when the list size increases.

4.3 Readset and writeset, reads and writes

A subtlety to clarify is the issue with expressing the size of the readset and the writeset. It is very common for these two sets to intersect, for which we have accounted inside the writeset; the readset measurements only include those transactional variables that are strictly only read. For example, in the case of SingleInt the readset size is always zero since the read element is in all the cases written.

All of the programs can be categorized using the number of times they perform a transactional read (marked “reads”) on committed transactions. In fact, the readset and the writeset are well-defined by the program itself and not by the transactional management.

Figure 8 shows the number of reads and the readset size, making it clear that `unreadTVar` functionality causes a very small commit readset (with no other changes whatsoever in reads/writes/writeset). As it is explained in [25], the proper use of `unreadTVar` on the linked list and binary trees

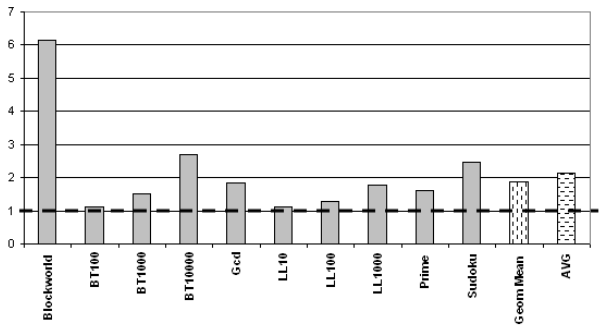

Figure 10: Avg. readset size of aborted transactions per avg. size of committed transactions (32 cores)

binds the readset of an insertion or a deletion to a maximum of three transactional variables. It makes the readset size contant and independent of the structure size, whereas the regular lists and trees have readset sizes as a function of their length.

In terms of efficiency regarding the writes, the optimal case is when the writeset equals the number of writes performed because the transactions only persist their updates by committing, and thus there would be no point in writing a transactional variable more than once. Figure 9 compares the number of writes to writeset size, showing that on this benchmark, the transactional variables are used efficiently (by producing similar numbers for each experiment). In case the STM programmer needs to change a value several times within a transaction, she can always do it with other kinds of variables (preferably with the ones with cheaper access, because STM writes imply identifying a variable in the writeset and only later updating the local copy), rather than with the Haskell `TVar`. Note that the final result do need to be in a `TVar` (therefore in the readset) to be committed.

Another interesting observation concerns the relation between the readset size and the rollback probability of transactions. Since an application can have transactions with different readset sizes, the intuition then is that the bigger the readset, the more probability that the transaction will rollback. Figure 10 confirms this intuition by showing the ratio between the average readset size of aborted transactions and the average readset size of the committed ones. It is sufficient that the plot only includes results of the 32 core executions since they have, in most cases, the greatest rollback rate. Only 10 out of the 18 applications in the suite are plotted because UnionFind does not rollback in a 32 core configuration, SingleInt does not have transactions with variable readset length and, as explained above, `unreadTVar` bounds the readset size of the transactions. As it could be seen, there is no value smaller than one which means that the rolled back transactions have, on average, the same or a bigger readset size than the committed ones. Ultimately, the plotted geometric mean and average, having values of 1,87 and 2,13 respectively, confirm the correlation between the readset size and the abort probability.

Another STM notion is the reads/writes ratio of the program, which is useful to see whether the program is read-dominated or write-dominated. On committed transactions, the SingleInt has a reads/writes ratio of 1, i.e. a read for each write performed. For the rest of the programs, this ratio is always much greater: on average, 99.1% of the transactional accesses are reads; which supports the fact that most applications are read-dominated by nature.

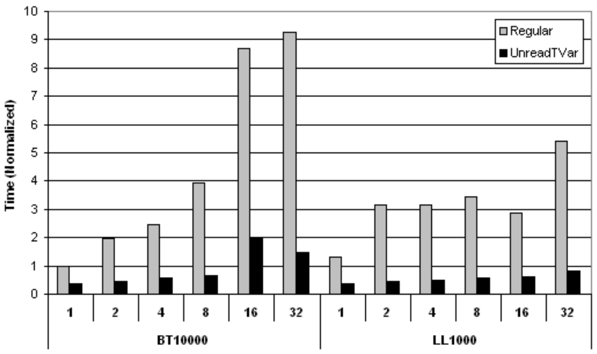

Figure 11: Commit time comparison between regular and `unreadTVar`-optimized LLs and BTs

4.4 Wasted and useful work

In a rollback situation, all of the speculative operations performed by the transaction have to be discarded and then re-executed; the time that the program has been executing is then, wasted. By dividing the wasted work by the sum of wasted and useful work, the percentage of overall wasted work is gathered. Figure 12 plots the percentages of wasted work per application, which is quite high for many cases, revealing another opportunity for future research.

Here is the proof that a profiling tool such as `readTStats` can be useful for researchers when reasoning about the findings and for measuring the impact of their ideas in the detail that they need, i.e. not only how faster the application is after applying the idea, but precisely the breakdown of the performance gain by transactional attributes. For example, it has been observed on Figure 2 that utilizing `unreadTVar` on a program that traverses linked structures helps with performance and scalability. On [25], it is hypothesized that the causes of this important speedup can be the reduction of two factors: the wasted work and the commit phase duration. Figure 12 shows that `unreadTVar` indeed helps to have much fewer rollbacks (and consequently less wasted work) so the first part of the hypothesis is confirmed. Moreover, Figure 11 shows that the commit times are much lower when `unreadTVar` is used, confirming the previous predictions on the causes of the speedup.

We believe that wasted work is a useful metric to indicate how well the applications scale transactionally: this could also help to decide the switchover point from transactions to acquiring a lock at the beginning of the atomic block. This approach attempts to avoid too much wasted work in the same fashion as speculative lock elision [22].

5. SCALABILITY ISSUES AND HARDWARE COUNTERS

The Figures 2 and 3 in Section 3 have showed that on the given architecture, most of the programs in the benchmark do not scale well for more than 8 cores. We analyze this issue given that it could be due to (1) lack of concurrency in the application, (2) the implementation of TM, (3) the Runtime and the Garbage Collection (GC) or (4) the system architecture.

The lack of concurrency of SingleInt is apparent, whereas “the CCHRs” were designed with small transactions to decrease the probability of conflict and to perform better running concurrently. The linked structures LL and the BT, especially with a large size and the `unreadTVar` optimization, should be perfect candidates for scalability by allowing

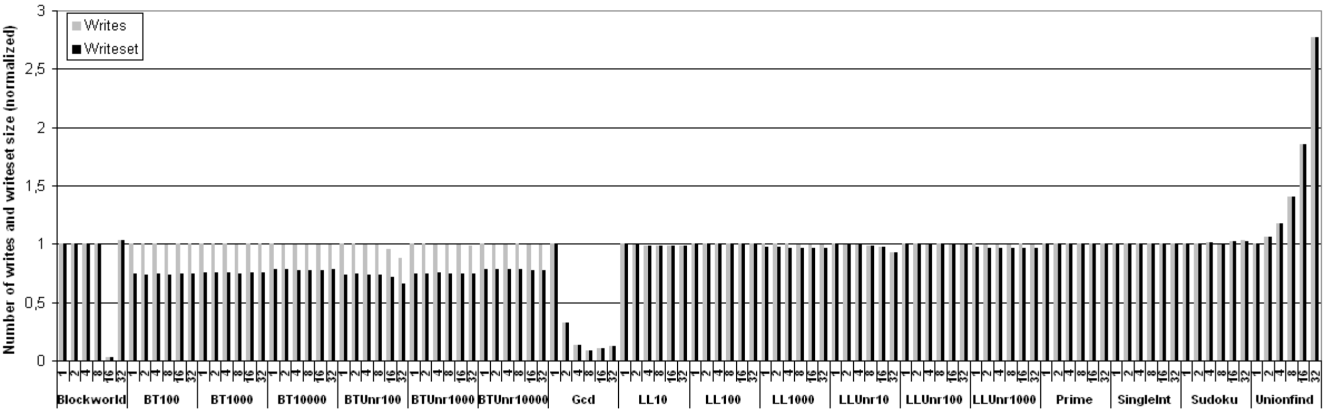

Figure 9: Number of writes and size of the writeset on committed transactions

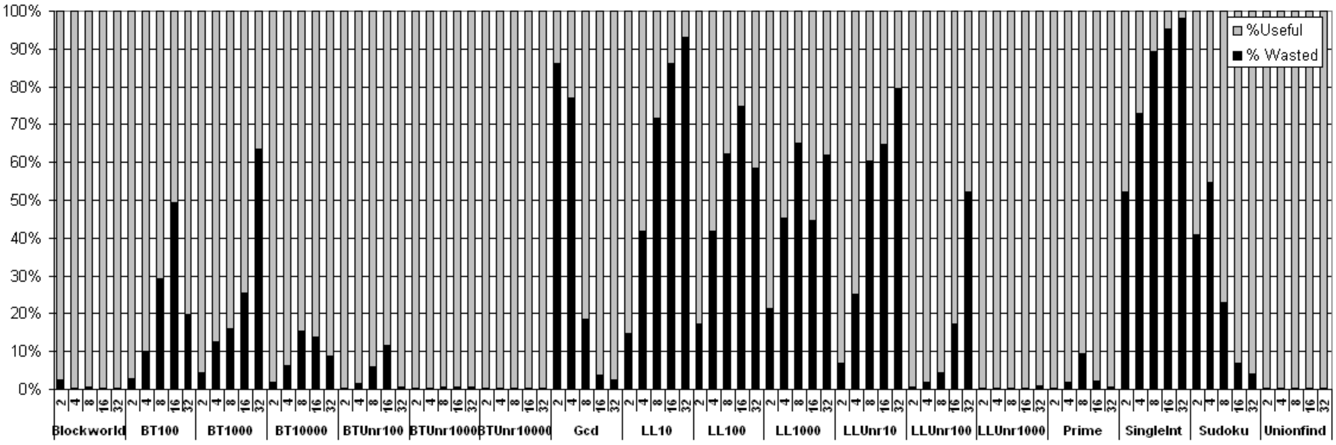

Figure 12: Percentage of wasted work per application

updates to distinct parts of the structure in parallel.

The biggest problem related to the specific Haskell implementation of STM that affects the scalability of the applications is the linearization point in the commit phase during writeback. Although the course-grained locking in the commit phase reduces the synchronization overhead to a single lock, it signifies a bottleneck, since it takes away the possibility of committing non-conflicting transactions in parallel.

Regarding the RTS and GC, it has been shown on [20] that the GHC runtime causes high allocation rates that result in high memory use and high transfer rates between the heap and the stack that unnecessarily occupy the functional units of the processors. Furthermore, its execution mechanism has a negative effect on branch predictors because the target addresses of jumps change frequently. Serialized garbage collection (GC) and lazy evaluation (which "may tend to hang on to the data longer than strict languages") are some of the factors affecting the performance in GHC negatively on modern processors.

As stated earlier, the architecture where the experiments were run is Non-Uniform Memory Access (NUMA). There are actually 5 different costs associated with inter-thread communication (writing to shared memory) depending on where the writer and the reader are scheduled. Two threads can be scheduled 1) on the same core, 2) on different cores on the same die, 3) on different dies on the same board, 4) on different boards on the same rack, 5) on different racks. Given that all the possible combinations might be present in the reported results except for the first case, the scheduling of transactional multithreaded applications seems to be the a promising research topic

Now we can examine the results obtained by looking into hardware counters of the Montecito [18] architecture.

5.1 The cache performance of the benchmark

Figure 7 shows that the overall transactional time (for committed transactions) almost always increases with the number of cores. That is to say that the commit phase takes more and more time, which is one of the factors that limit scalability.

Figures 13 and 14 show the percentage of L1 misses for the commit phase and the work phase, respectively. In our benchmark, for transactions that end up committing, there are on average 27% percent L1D, and 4% L2D cache misses (not shown). For aborts, there is more variation, depending on how early the abort took place inside the transaction. For programs that have a large readset (such as BT100, BT1000, BT10000, LL100 and LL1000), at the writeback phase of the commit, the caches are "hot" since the dataset has already been traversed once in the validation phase, and therefore the transactions that end up committing have a lower L1 cache miss rate than transactions that end up aborting.

In the Haskell RTS, the readset and writeset are implemented together using a linked list of so-called "chunks". Each chunk is an array (good cache locality) that is chained via a pointer (bad cache locality) to another chunk. In this work, a chunk size of 16 was used, which is the default size that GHC defines. Figure 14 shows the way this implementation affects the cache miss ratio of those applications that repetitively access these structures (i.e. LL and BT).

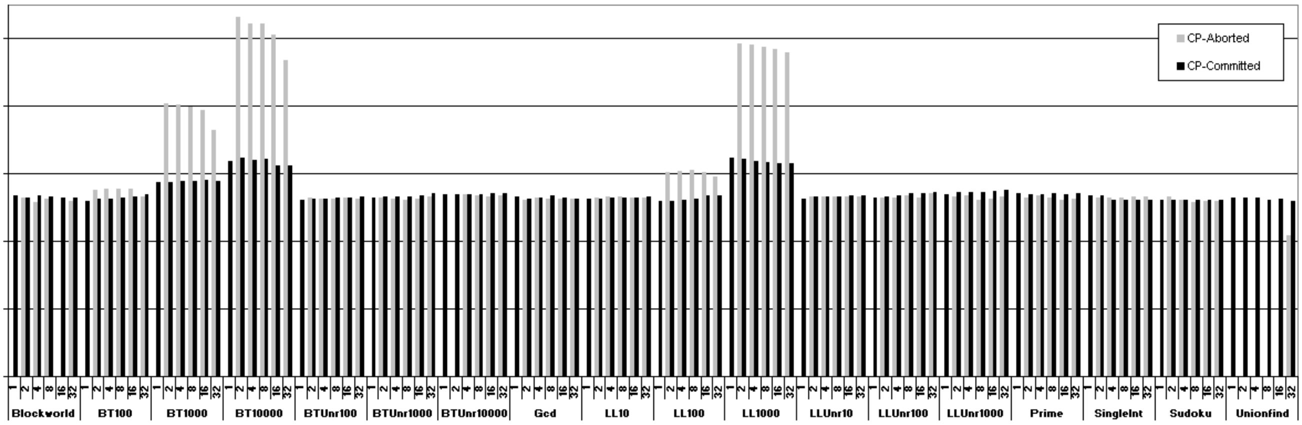

Figure 13: L1D misses (Commit Phase)

As a result, large L1D cache miss ratios are observed in the work phase for large linked structures, such as BT10000 and LL1000, as well as small miss ratios for applications with a fittable-in-cache readset such as the ones using `unreadTVar`. Notice that even though the readset and writeset size of SingleInt is equal to one in all the cases, there is no cache locality to take advantage of since the dataset is accessed a couple of times and then the transaction finishes, getting rid of all the data structures that it has been using.

5.2 Cache utilization per transactional access

Another useful measure would be to look at the work phase of the transactions that end up committing. The metric of cache accesses per transactional access (read and write set combined) might give us a rough idea of the necessary accesses to data while processing a transaction. For this, Figure 15 presents L2 data cache accesses per committed transaction. Of course, the data accesses made by the Haskell program inside a transaction also contain purely functional work, as well as calls made by the RTS, per-thread transaction record management, scheduling, GC, etc. The applications on our benchmark show tendencies to show results close to a per-application average; so that they can be grouped in two:

- High cache utilization: The CCHR applications and SingleInt. At first, it might seem very counter-intuitive that the SingleInt benchmark, having a single atomic block that only updates a variable, performs around 1000 cache accesses per each transactional access. Nevertheless, as explained in the previous subsection, what actually happens is that since a new transaction is created every time that the shared counter has to be incremented, the overhead of book-keeping the data structures needed to maintain transactions and validating them is enormous. CCHR performs similarly with its many short transactions.
- Low cache utilization: Using the same criteria as above, since LL and BT mostly perform transactional accesses and their transactions are longer, these applications have fewer cache accesses per transactional access. In spite of that, the ratios are still large even in this group, showing that the work the RTS has to do in order to maintain transactional structures is quite high.

5.3 Stalls

The data cache misses on GHC impact the performance so negatively that the time lost in commit phase due to stalls easily cross the 50% mark (Figure 16). Moreover, most of the stalls are due to caches. In Figure 16, the number of stalls are much higher with increasing core count, indicating the “ping-ponging” of cache lines. Furthermore, the number of stalls are higher for the commit phase than for the work phase; since in the commit phase GHC’s runtime compares its private copy with the original data residing in main memory or in another cache. One might be tempted to ask if some of the stalls could be due to branch mispredictions. The answer is no: in examining the data in Figure 17, which shows the distribution of branch mispredictions, and comparing it with Figure 16, we see that the two graphs are very weakly correlated, i.e. a rise in branch misprediction rate does not necessarily impact the percentage of time spent in stalls in a similar fashion. Please also note that branch misprediction rate is lower for the commit phase since the control flow is more deterministic, dominated by loops.

These results partially agree with the findings in [20] for non-transactional Haskell applications. In that work, L2 data cache misses were accounting for up to 60% of the execution time, with a 32% branch misprediction rate on average. For Haskell STM applications, we also observe a very high data cache miss rate, but the branch misprediction rate is lower. In comparison with the benchmark on [20], Haskell STM programs were observed to be effected even more negatively by the low cache performance than for non-transactional programs.

6. CONCLUSIONS AND FUTURE WORK

In this work, the transactional behavior of several Haskell STM programs was analyzed. One of the distinguishing factors of STM applications, the rollback rate, can help determine the transactional scalability of the application. Especially when different threads constantly update the same variables, there is no way of making the critical section parallel, so it is intuitively expected to have a big number of rollbacks. On these cases, where some transactions rollback many times, it might be necessary to dynamically tune the runtime system in order to avoid excessively aborting transactions. Commit phase overhead is another useful metric that we propose, which can be used to evaluate STM performances. This work also showed that a bigger readset size is correlated with a greater probability of rollback.

In a nutshell, although providing the application with atomic safety and additional cores might appear to be promising to increase performance, the overhead associated with the transactional management (and therefore the average

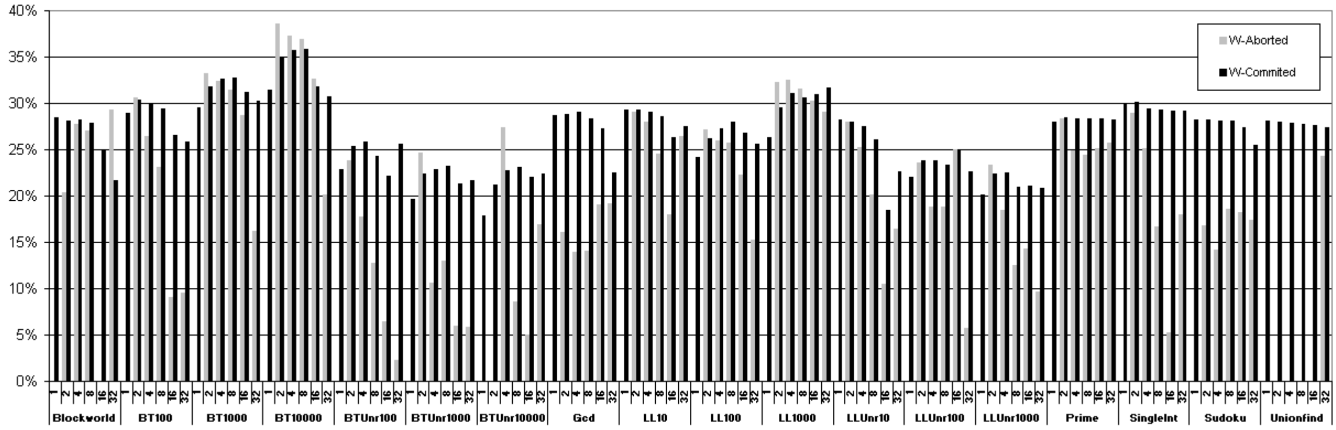

Figure 14: L1D misses (Work Phase)

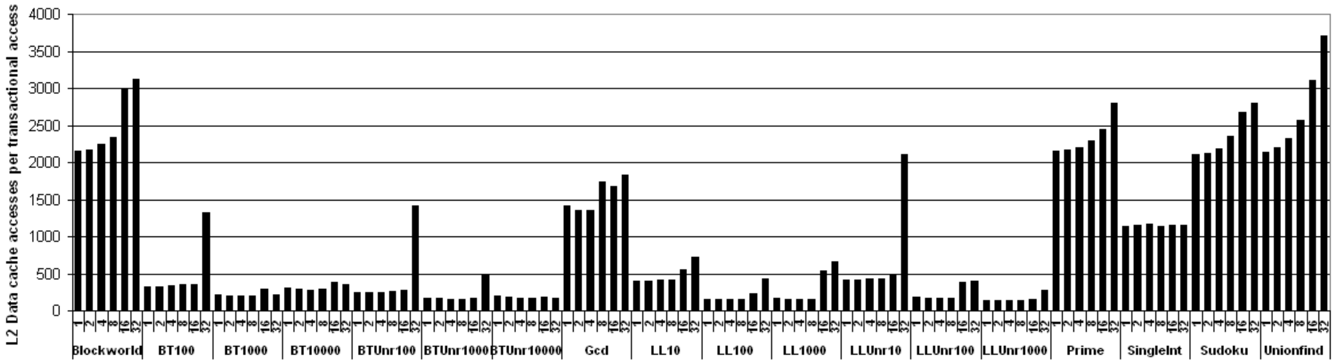

Figure 15: L2D accesses per each transactional access

transactional time) also increases. Running our benchmark on the Altix 4700 ccNUMA machine, most of the applications and the Haskell STM system of GHC runtime undergo serious scalability problems, especially working with more than 8 cores. Revising the GHC runtime's execution mechanism, trying to optimize the thread scheduling for STM execution, parallelizing the GC and enabling concurrent commits could be some of the ways to improve the performance of Haskell STM in the future. The bottlenecks can be identified by using the extensive analysis of each program's transactional behavior; their abort rates, commit phase overheads and details about transactional accesses and hardware performance counters provided by the instrumentation methodology developed for this work.

In particular, for programs that do not perform much work inside transactions, the commit overhead appears to be very high. To further observe this overhead, an analysis needs to be conducted on the performance of commit-time course-grain and fine-grain STM locking mechanisms. Fine-grained commit phase locking is another option on the Glasgow Haskell Compiler, the possibility of choosing the most suitable granularity per application is left as future work. Additionally, more future research needs to tackle the issue of whether the current system architecture together with the STM runtime conform to the demands of transactional management. This is mandatory to have STM be more applicable in the many-core future.

More research into techniques such as `unreadTVar` that give programmers the possibility of reaching a better performance have to be carried out to brake the limits found in this work. Moreover, hardware mechanisms to help Haskell RTS manage transactions in a more efficient way should also be explored.

Acknowledgements

The authors would like to the Gelato community, Philip Mucci, Oriol Prat and Roberto Gioiosa for all the useful comments and advice.

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 and by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC).

7. REFERENCES

[1] Haskell official site: http://www.haskell.org.

[2] Sgi numalink interconnect fabric: http://www.sgi.com/products/servers/altix/numalink.html.

[3] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng, and D. Yeung. Biobench: A benchmark suite of bioinformatics applications. In *Procs. of ISPASS 2005*. Mar 2005.

[4] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. *ISCA*, pages 81–91. ACM, 2007.

[5] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.

[6] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional

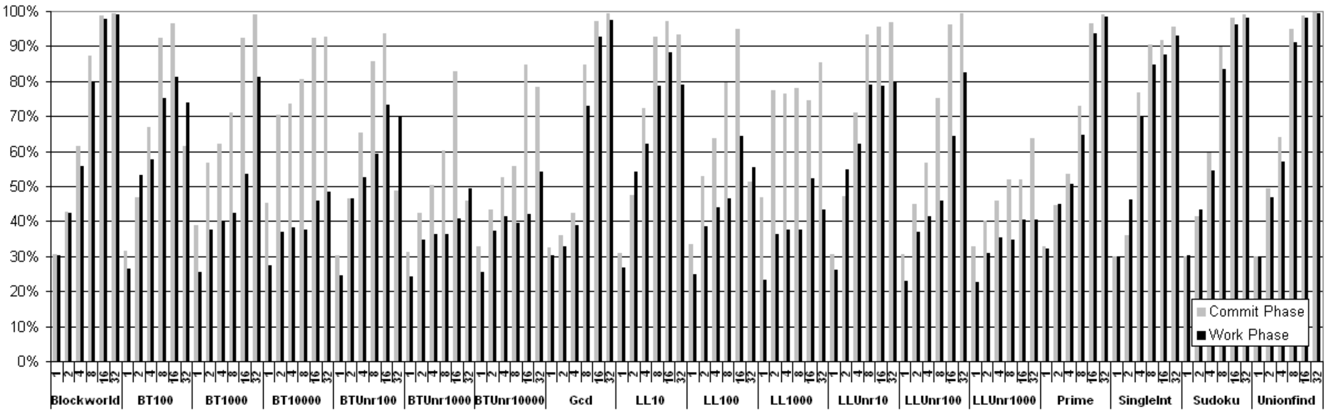

Figure 16: Percentage of time in stalls in committed transactions

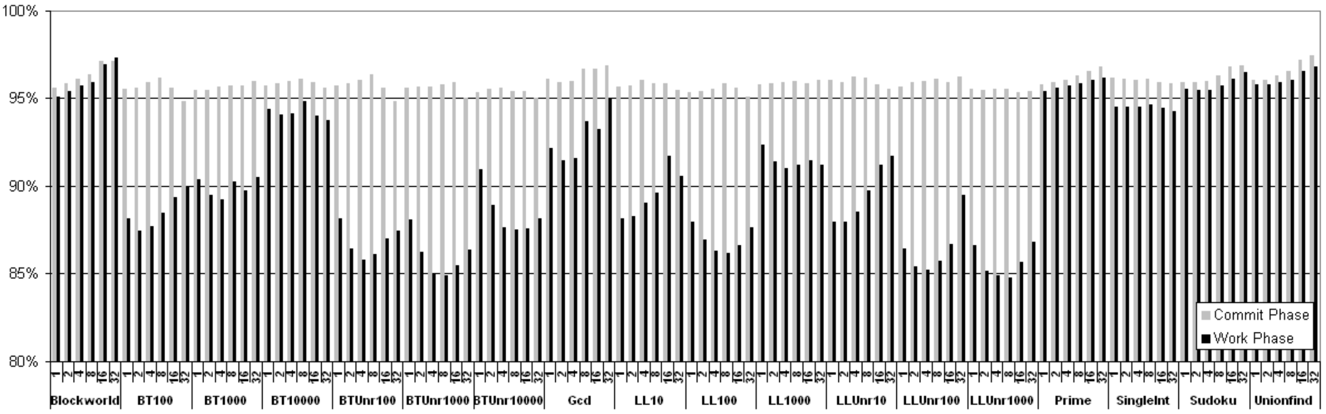

Figure 17: Completed/issued instructions for committed transactions

memory system with strong isolation guarantees. In *Procs. of the 34th ISCA*. Jun 2007.

[7] J. Chung, H. Chafi, C. Cao Minh, A. McDonald, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. The common case transactional behavior of multithreaded programs. In *12th HPCA*. Feb 2006.

[8] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Procs. of the 12th ASPLOS*, pages 336–346, 2006.

[9] H. Daume III. Yet another haskell tutorial. In *www.cs.utah.edu/ hal/docs/daume02yaht.pdf.* 2002-2006.

[10] T. Fruhwirth. Parallelizing union-find in constraint handling rules using confluence. In *21st Conference on Logic Programming ICLP*. Oct 2005.

[11] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: A benchmark for software transactional memory. In *Procs. of EuroSys2007*, 315–324. ACM, Mar 2007.

[12] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *PPoPP'05*, Chicago, Illinois, June 2005.

[13] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Procs. of the Twentieth Annual ISCA*, 1993.

[14] S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23^{rd} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.

[15] E. S. L. Lam and M. Sulzmann. A concurrent constraint handling rules implementation in haskell with software transactional memory. In *DAMP'07.* Jan 2007.

[16] J. R. Larus and R. Rajwar. *Transactional Memory.* Morgan and Claypool, 2006.

[17] L. Man-Lap, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The alpbench benchmark suite for complex multimedia applications. In *Procs. of IISWC-2005.* Oct 2005.

[18] C. McNairy and R. Bhatia. Montecito: A dual-core, dual-thread itanium processor. *IEEE Micro*, 25(2):10–20, 2005.

[19] R. Narayanan, B. Özisikyilmaz, J. Zambreno, G. Memik, and A. N. Choudhary. Minebench: A benchmark suite for data mining workloads. In *IISWC*, pages 182–188. IEEE, 2006.

[20] N. Nethercote and A. Mycroft. The cache behaviour of large lazy functional programs on stock hardware. In *Procs. of the ACM SIGPLAN Workshop on Memory System Performance.* ACM Press, 2002.

[21] W. Partain. The nofib benchmark suite of haskell programs. In *Workshops in Computing, Springer Verlag.* 1993.

[22] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Procs. of the 34th International Symposium on Microarchitecture*, 294–305. Dec 2001.

[23] N. Shavit and D. Touitou. Software transactional memory. In *Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[24] T. Skare and C. Kozyrakis. Early release: Friend or foe? In *Workshop on Transactional Memory Workloads.* Jun 2006.

[25] N. Sonmez, C. Perfumo, S. Stipic, A. Cristal, O. Unsal., and M. Valero. Unreadtvar: Extending haskell software transactional memory for performance. In *TFP 2007.* Apr 2007.