Dissecting Transactional Executions in Haskell

Cristian Perfumo[+*] Nehir Sonmez[+*] Adrian Cristal[+]
Osman S. Unsal[+] Mateo Valero[+*] Tim Harris[#]

[+]Barcelona Supercomputing Center,
[*]Computer Architecture Department, Universitat Politècnica de Catalunya, Barcelona, Spain
[#]Microsoft Research Cambridge
{cristian.perfumo, nehir.sonmez, adrian.cristal, osman.unsal, mateo.valero}@bsc.es tharris@microsoft.com

Abstract

In this paper, we present a Haskell Transactional Memory benchmark in order 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. Using a composite of the collected raw data, we propose new transactional performance metrics. We analyze key statistics relative to critical regions, transactional log-keeping and overall transactional overhead and accordingly draw conclusions on the results of our extensive analysis of the set of applications. The results advance our comprehension on different characteristics of applications under the transactional management of the pure functional programming language, Haskell.

Keywords Transactional Memory, Multicores, Haskell, Concurrent Programming, Transactional Benchmarks

1. Introduction

Research on Transactional Memory (TM) has been done over more than two decades with different flavors, semantics and implementations [1, 2, 3]. Even if a lot of intuitive conclusions can be made after observing execution times and speedup analysis, it might be useful to look deeper inside a transactional application execution to extensively inspect the most relevant characteristics of TM, such as number of committed transactions, rate of aborts, read and 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 some results to eventually make something faster, better or easier.

As hinted above, in the particular case of Transactional Memory, some questions arise when an application is executed: Does my application suffer from a 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? The aim of this work is to profile a set of transactional Haskell applications and to draw conclusions out of the data obtained from monitoring actual applications. In order to accomplish this, the Haskell Runtime System (RTS) has been modified by inserting monitoring probes. Those probes extract application runtime data relevant to the software transactional execution.

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 [4, 5, 6], none of the applications in those benchmarks use Transactional Memory. Pre-TM era Haskell benchmarks exist [7], and recently a one-application highly-tunable TM benchmark was developed for imperative languages: STMBench7 [8]. No TM benchmark exists for Haskell; an appropiate environment to test new TM ideas given its small, simple, and easy-to-modify runtime system. This work addresses this issue.

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 to collect transactional information such as commit and abort rates, as well as their runtime overheads on those applications.
- Based on the collected raw transactional information, new metrics such as wasted time and useful work are derived, which could be used to characterize STM applications. For example, it is observed that applications can be classified into subclasses such as low, medium and high abort-rate, based on clustering the harvested runtime transactional information. 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 suite, section 4 discusses the statistics obtained from the instrumentation of the runtime system and finally section 5 concludes and looks into future work.

2. Background: Transactional Memory in Haskell

The Glasgow Haskell Compiler 6.6 (GHC) [9] provides a compilation and runtime system for Haskell 98 [10], a pure, lazy, functional programming language. The GHC natively contains STM functions built into the Concurrent Haskell library [11], 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) forces threads to access shared variables only inside a transaction. This useful restriction is more likely to be voilated 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]. STM-Haskell has the attractions that (i) the runtime system is small, making it easy to make experimental modifications, (ii) 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 STM-Haskell implementation.

STM provides a safe way of accessing shared variables between concurrently running threads through the use of monads [1], having 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), whereas only I/O actions and pure computations, and not STM actions can be performed outside a transaction. This guarantees that `TVars` cannot be modified without the protection of `atomically`. This kind of protection is well known as "strong atomicity"[13]. Moreover, in the context of Haskell and due to monads, the computations that have side-effects from the ones that are effect-free are completely separated. Utilizing a purely-declarative language for TM also provides explicit read/writes from/to mutable cells; memory operations that are also performed by functional computations are never tracked by STM unnecessarily, since they never need to be rolled back [12].

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

Table 1. Haskell STM Operations

Threads in STM Haskell communicate by reading and writing transactional variables, or `TVars`. All STM operations make use of the `STM` monad, which supports a set of transactional operations, including allocating, reading and writing transactional variables, namely the functions `newTVar`, `readTVar` and `writeTVar`, respectively, as it can be seen on Table 1.

Transactions in Haskell are started within the `IO` monad by means of the `atomically` construct. 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 the system 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 `TVars` involved, making these effects visible to other transactions. This method deals with maintaining a per-thread transaction log that records the tentative accesses made to `TVars`. When `atomically` is invoked, the STM runtime checks that these accesses are valid and that no concurrent transaction has committed conflicting updates. In case the validation turns out to be successful, then the modifications are committed altogether to the heap.

3. Analyzed Applications

The transactional analysis contains 13 applications, some of which have been written by people who are not necessarily experts on implementation details of Software Transactional Memory, whereas others have been developed by the authors of this work who have worked using and modifying Haskell STM runtime for some time. Both kinds of applications are interesting because the former can show the way regular programmers (the ultimate target of the whole TM research) use Transactional Memory 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 that are different from the common ones.

In the set of applications itemized in Table 2, all the runs are arranged according to the execution environment to fit

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, the nodes are the elements, and the roots are the representative of the sets (CCHR)	1157	13
TCache	A shopping simulation of 200000 purchases performed by as many users as the threads the program has. It updates stock and spent money per customer. The information is maintained using a library called Transactional Cache [14]	315	6
SingleInt	A corner-case program that consists of n threads incrementing a shared integer variable for a total of 200000 times. Therefore, the access to the critical section is serialized in this application because the parallelization of two 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` [15], a performance improvement that uses early release [16], i.e. 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

Table 2. Description of the applications in the benchmark, number of lines of code and atomic blocks

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 [17], where the basic idea is to derive a set of rules and applying 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 union find 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 [18]. The UnionFind application provided in the implementation of CCHR simply generates a several disjoint sets of integers and combines them all together.

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. This one is the output of the application. The "active" rules are maintained in a so called "execution stack". Active rules search for possible matching rules in order and they become inactive when all matching rules have been executed or the constraint is deleted from the store.

In order to allow thread-level paralellism, several "execution stacks" are simultaneously mantained, meaning that multiple active constraints can be evaluated at the same time. The applications also balance the workload by moving constraints from one execution stack to another.

TCache is a program that uses a library called Transactional Cache [14] as a base. The application included in the benchmark models a shop with several clients purchasing items available in stock. The program finalizes when there are no more items to sell. The multi-threaded versions has as many buyers as threads in the application and the shop starts with 200000 for-sale items. Information about the purchases is also stored in text files.

SingleInt is a simple program that updates an integer variable that is shared among the threads. Both SingleInt and TCache are programs that try to perform in parallel a task that is inherently sequential. Since they update the shared structure (integer and cache respectively) and almost no further computation is carried out, their 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. The workload is equally divided among the threads and always totalizes 200000 operations. Lists of different lengths (10, 100, 1000) were analyzed given that the behaviour is intuitively expected to be different. Two flavors of linked lists exist in the benchmark: the regular, and the unreadTVar-optimized [15]. The next node link of the linked list is implemented as a transactional variable. 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 A first-order runtime analysis

Before starting to examine detailed internal transactional behavior of the applications, it is important to have a global idea about their execution time and the percentage of this time that each application spent inside a transaction. Table 3 shows how long, in seconds, each program ran for 1, 2, 4 and 8 cores versions. Figure 1 plots these values normalized to the single-core version and so, comparisons among the scalability of different programs are straightforwardly visible. Most of the applications have smaller runtimes as the number of threads goes up but there are a few exeptions that

Application	1 core	2 cores	4 cores	8 cores
Blockworld	13.43	7.30	13.24	5.91
Gcd	76.83	28.78	5.40	2.35
LL10	0.08	0.08	0.07	0.26
LL100	0.31	0.28	0.25	0.25
LL1000	4.84	3.77	3.72	3.19
LLUnr10	0.08	0.08	0.06	0.21
LLUnr100	0.36	0.24	0.18	0.24
LLUnr1000	3.10	1.92	1.16	0.83
Prime	38.41	21.19	14.14	8.44
SingleInt	0.12	0.52	0.62	0.77
Sudoku	0.72	0.44	0.40	0.32
TCache	2.58	3.52	4.20	4.70
UnionFind	2.64	1.78	1.17	0.74

Table 3. Execution time (secs) of each application

are worth analyzing: SingleInt and TCache experience high contention because the task they perform does not match the optimistic approach of most transactional memory systems (including Haskell). These applications were explicitly written to conflict and so, the more the conflicts, the more the rollbacks, the more the re-executed tasks, the more the cache thrashing and the more the spent time. The other two applications that suffer from performance degradation are both versions of linked list (LL and LLUnr), specially with 8 cores. The explanation for this phenomenon is that the number of elements in the list is almost the same as the number of cores, increasing the number of conflicts.

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. For this reason we included Figure 2 that shows the percentage of time each application is inside a transaction for all its different core counts. As it can be seen in this figure, several applications spend a substantial amount of time in running transactions, therefore improvements in the transactional runtime system will non-marginally reduce the overall execution time.

4. Observed Statistics

All the experiments are run in a four dual-core processor SMP server with Intel Xeon 5000 processors running at 3.0 GHz with 4MB L2 cache/processor and 16GB of total memory, and all of the reported results are based on the average of five executions.

The statistics accumulated by the proposed extension to the runtime system are encapsulated in a function called `readTStats`, which by being invoked retrieves the whole list of raw statistics shown in Table 4. This function resides in the file that contains all STM functions (`STM.c`) inside the Glasgow Haskell Compiler (GHC) 6.6, where the values are gathered at different points in the transaction execution, i.e. when 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 new fields were added to the `TRec` structure. Although this procedure does add some overhead on the transactions, it is the unique way of achieving our objective in this context. Information about timing is obtained by querying hardware performance counters. 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 commit attempt.

Out of these values we derive other 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. Note that these derived statistics such as wasted work could be used as metrics to gauge the transactional performance of applications using STM.

In our experiments, the total STM execution time (the time spent executing inside an atomic block) is divided into two: the time spent for doing the work the transaction is supposed to do (including the transaction start and the transactional book-keeping), and the commit phase itself, which signifies the time taken for committing or discarding the tentative changes. The start transaction phase, which is always constant and negligable 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 will be observed, such as the commit phase overhead, data related to the transactional record, and the percentage of wasted work.

4.1 Abort Rate

To start off, it is important to state that throughout this work, the terms abort and rollback are used interchangably. However, the reader should be aware that in other contexts 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 3) in a multicore environment, a classification was made based on the observed values:

- High abort rate group: In this first group of applications reside the SingleInt and the TCache. These programs continuously update their shared data, therefore abort most of the time on multicore and also suffer from high cache thrashing.
- Medium abort rate group: LL10, LL100, LL1000 and LLunr10 abort frequently. 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 almost ending up equaling the number of cores present. Even for larger lists, the significantly larger

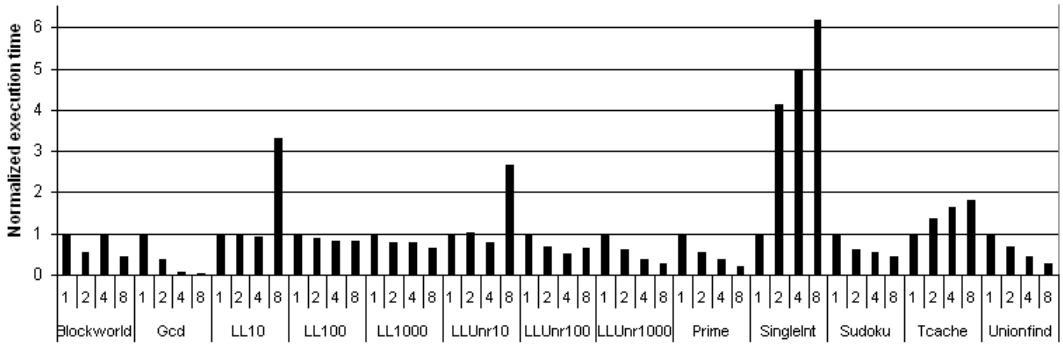

Figure 1. Execution time normalized to the single-core time

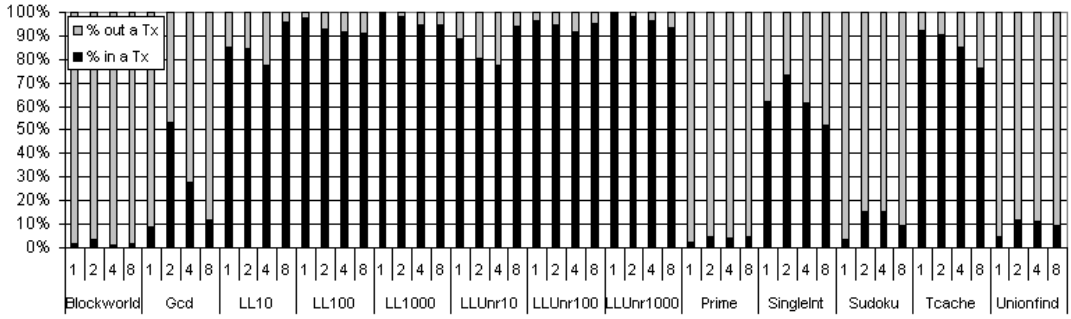

Figure 2. Percentage of the time the application is inside and outside a transaction

number of aborts is a general case for linked structures since the STM tries to look after more than what is needed by collecting all traversed elements in the readset [15].

- Low abort rate group: LLunr100 and GCD rarely abort, Blockworld, Sudoku, Prime, LLunr1000 almost never abort and UnionFind never aborts. The low rollback rate of LLunr100 and LLunr1000 is a consequence of the efficiency of `unreadTVar`. In general, CCHR applications do not rollback very often, and instead make use of the atomicity promise due to the fact that CHR naturally supports concurrent programming, and wants to make use of the simplicity of using transacted channels while doing so [17].

It would also be interesting to look at how the aborts are distributed. For this purpose, the histograms in Figure 4 show the distribution of rollbacks per transaction from 0 to 10+ times. It can be clearly seen that the transactions of the programs tend to be able to commit mostly in the first try, except for the cases of TCache, 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. Our findings show that for some applications certain transactions abort repeatedly sometimes more than 10 times, which shows the need for research into STM runtime mechanisms to avoid excessively aborting transactions for the sake of fairness.

4.2 Work time, commit time and commit phase overhead

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

Figure 5 illustrates the way that time is spent when the application is within a transaction, taking into account only committed transactions. 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 and how much to commit it. The commit phase overhead is obtained by dividing the commit phase time by the total time. We believe that commit phase overhead could be one of the appropriate metrics to measure STM implementation performance. The highest commit phase overhead per committed transaction is present in the programs belonging to the CCHR set: reaching almost 70% in the case of UnionFind running on two cores (GCD is an exception with no more than 24%). For scalability, these applications have a very large number of small transactions to avoid costly rollbacks produced by large transactions, impacting the overall performance negatively.

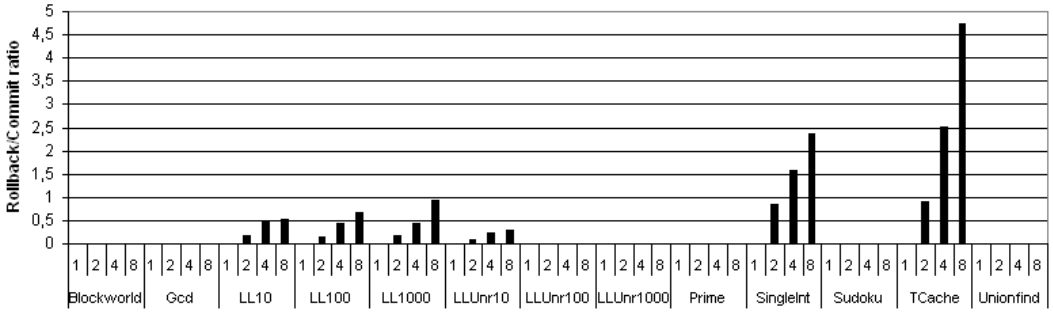

Figure 3. Rollback rate: Number of rollbacks per committed transaction

Statistics	Explanation
CommitPhaseTime	Accumulated time the application spent in the commit phase (transactions that have finally committed)
CommitNumber	Number of committed transactions
CommitReads	Total number of transactional reads in committed transactions
CommitReadset	Sum of the readset sizes of all committed transactions
CommitWrites	Total number of transactional writes in committed transactions
CommitWriteset	Sum of the writeset sizes of all committed transactions
CommitWorkTime	Accumulated time that the application spent inside a transaction that finally committed (useful transactional work)
AbortPhaseTime	Accumulated time the application spent in the commit phase (transactions that rolled back in the end)
AbortNumber	Number of rolled back transactions
AbortReads	Total number of transactional reads in rolled back transactions
AbortReadset	Sum of the readset sizes of all rolled back transactions
AbortWrites	Total number of transactional writes in rolled back transactions
AbortWriteset	Sum of the writeset sizes of all rolled back transactions
AbortWorkTime	Accumulated time that the application spent inside a transaction that finally aborted (wasted transactional work)
HistogramOfRollbacks	Number of transactions that were rolledback 0 .. 10 times and more than 10 times

Table 4. Summary of the statistics gathered by readTStats

Regarding high abort-rate applications , SingleInt (being second highest) had on average 25,1% commit overhead per committed transaction, while TCache only had 3,4%. The explanation for such a big difference between these two applications that belong to the same group is that for the single integer application, the only work that has to be done is to read and write back a variable, whereas in the latter, the relatively larger work involves finding the item, the user, and calculating and updating the stock values.

For the linked list programs, larger list size implies less commit overhead because although there is a larger readset and writeset and a greater number of reads in these programs, the commit time does not rise as rapidly as the work time, as the list size goes up.

Another observation is that when the number of cores increases, the number of aborts usually increases as well, except for BlockWorld and UnionFind that almost never abort (i.e. they abort either zero or once). Having more aborts when the number of cores (and then threads) goes up is due to the situation of having more concurrency and parallelism in the application. This increases the probability of conflict and, therefore, rollback.

A last interesting point is that average time per commit increases as number of cores goes up. Especially the biggest jump is observed when going from 1 to 2 cores, with a 2-4x increase. Although this needs further investigation, a sensible explanation for this is that shared variables are more likely to be spread among local caches of all the cores, and therefore, the situation mentioned above might be due to cache misses. Although architectural concerns are beyond the scope of this paper, this is an incidence where the architecture of the system affects the calculations. For aborts, there is only a slight increase in commit overhead because all that has to be done is to nullify local (i.e. speculative) changes.

4.3 Readset, writeset, reads, writes and reads/writes

Firstly, it should be pointed out that on committed transactions, the number of reads per transaction is constant for almost all programs regardless of the number of cores, since the readset and the writeset is usually defined by the program itself and not the transactional management (Figure 6). A subtlety to clarify is the issue with expressing the size of the readset and the writes. It is very common for these two sets to intersect, for which we have accounted for inside the writeset, rather than the readset. So our readset measurements only include those transactional variables that are strictly only read.

In terms of efficiency regarding the writes, the optimal case is that the writeset equals the number of writes performed because the transactions only persist their changes by committing, and thus there would be no point in writing

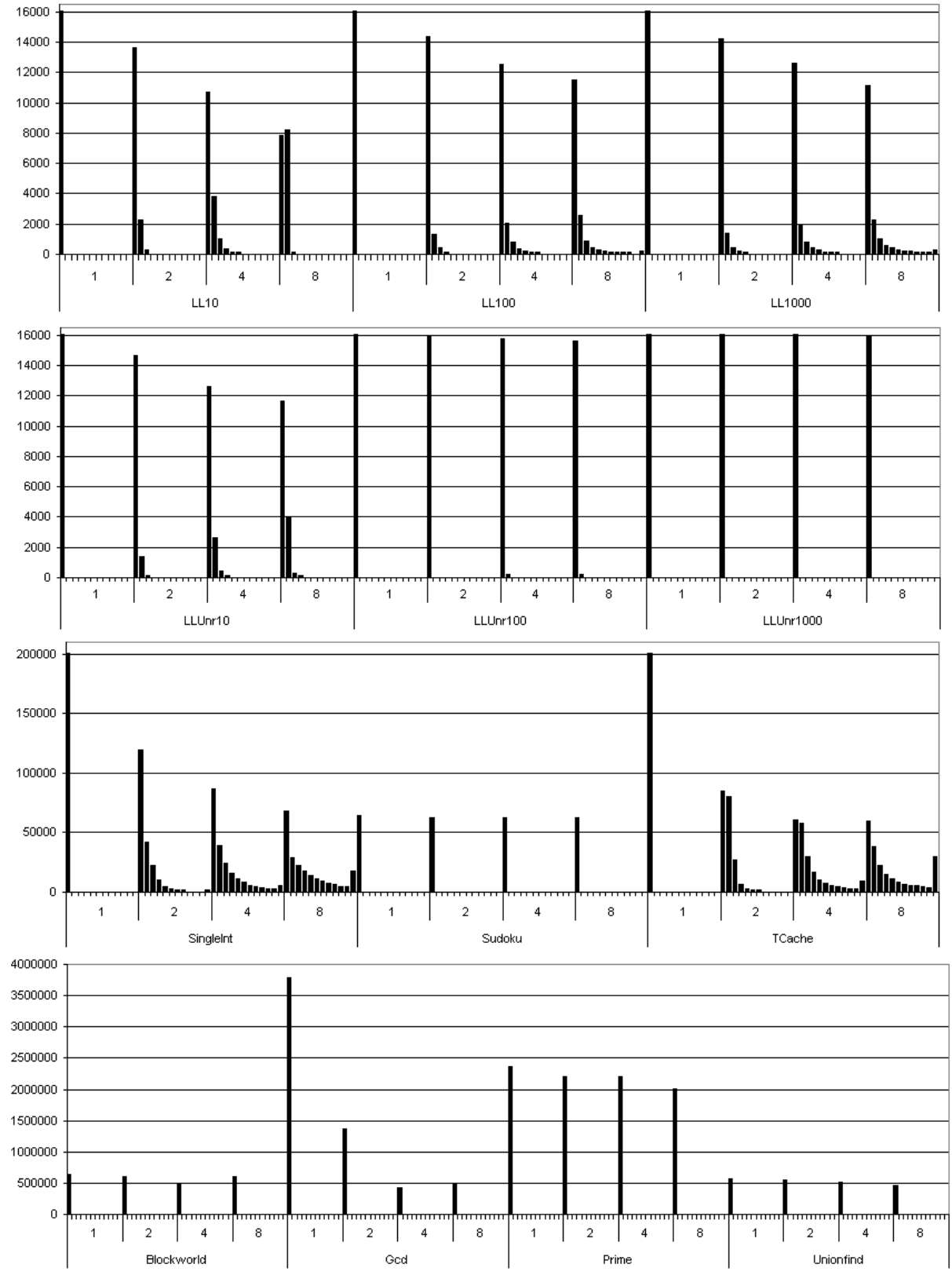

Figure 4. Rollback histograms

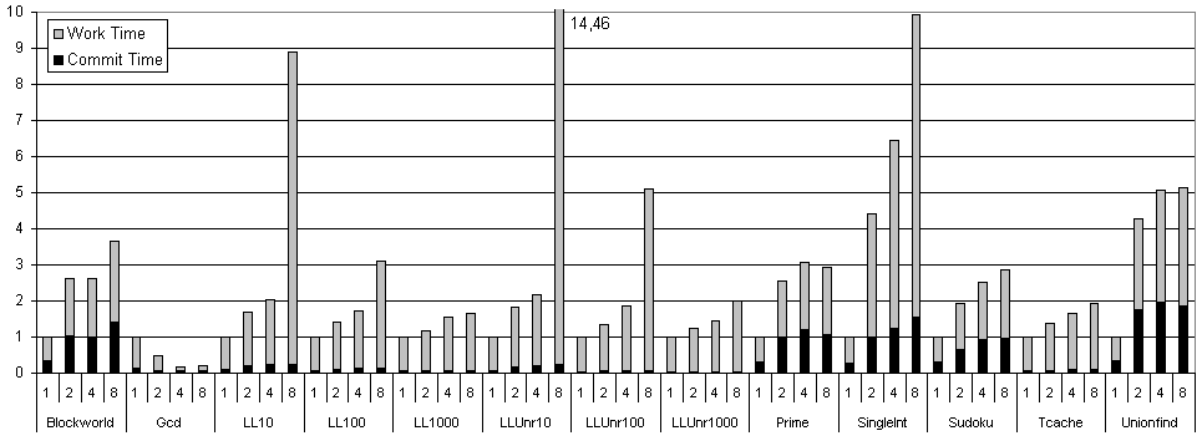

Figure 5. Commit phase overhead (Normalized to single core time)

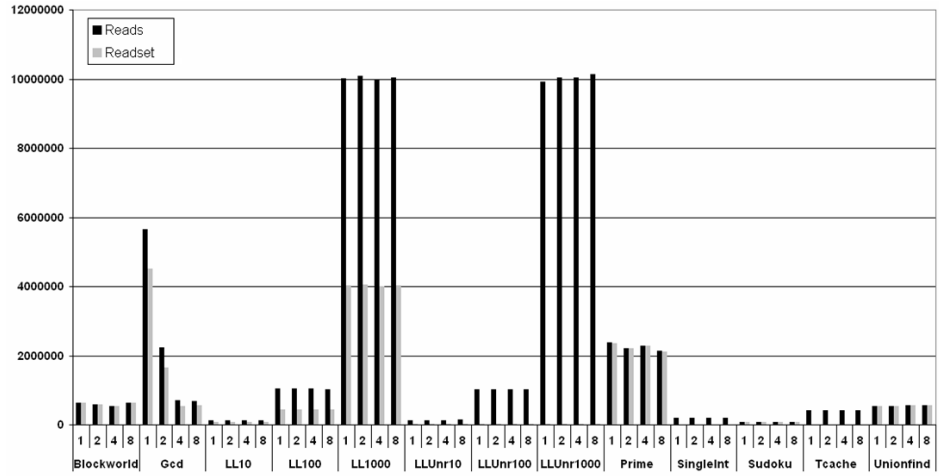

Figure 6. Number of reads and size of the readset on committed transactions

a transactional variable more than once. Figure 7 compares number of writes to writeset size, showing that programmers usually use transactional writes efficiently. In case the STM programmer needs to change a value several times within a transaction, she can always do it with other kind of variables rather than with the Haskell transactional variable TVar (ones with cheaper access, because STM writes imply identifying a variable in the writeset and only after that updating the local copy) and then have the final result in the writeset to be committed. This optimization could also be performed by the compiler.

In the case of the linked list (with and without `unreadTVar`) on commits, while the number of writes is constant for all list sizes, the reads increment about 10 times as list length is increased by an order of magnitude, which should be obvious because traversing a ten times bigger list leads to reading ten times more nodes on average.

`unreadTVar` functionality causes a very small commit readset (with no other changes whatsoever in reads/writes/writeset) and for aborted transactions it presents very small numbers for all values of reads, writes, readset, and the writeset. As it is explained in [15], the proper use of `unreadTVar` on the linked list binds the readset of an insertion or a deletion to a maximum of three transactional variables, and as seen on Figure 6, the readset size is contant and independent of the list size, whereas regular linked lists have readset sizes proportional to their length.

For the aborted linked list transactions (not shown), it was observed that all these four statistical values significantly decrease as more cores are introduced. This suggests that since there are less transactional variables to check for, aborts are less costly with more cores, at least for the commit phase. However the wasted work is still wasted and is usually increasing with more cores.

Another interesting observation concerns the relation between the readset size and the rollback probability of transactions. In the case of Haskell STM, a transaction T1 will be rolled back if and only if some other transaction Ti has modi-

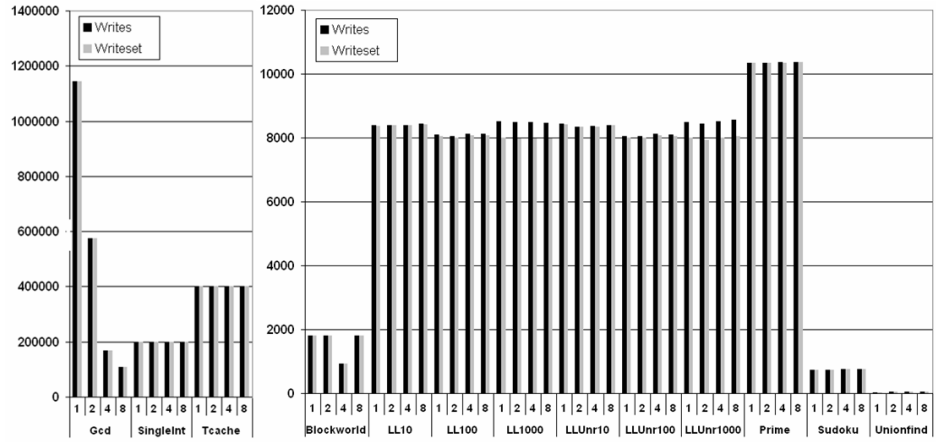

Figure 7. Number of writes and size of the writeset on committed transactions

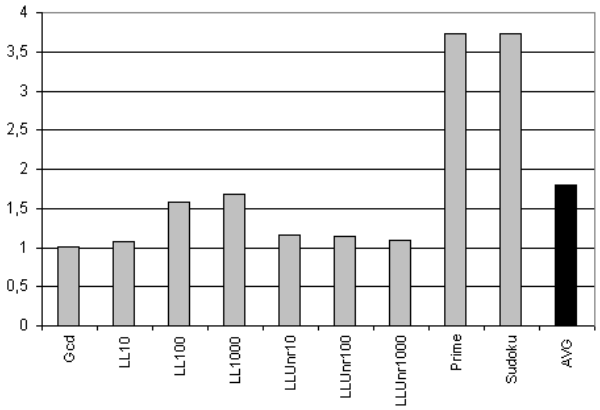

Figure 8. Average readset size of aborted transactions per average size of committed transactions (8 cores)

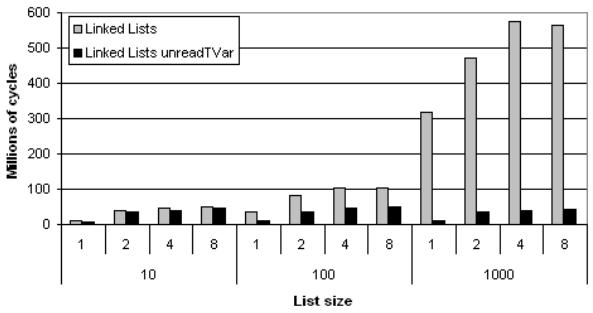

Figure 9. Commit time comparison between regular linked list traversal and `unreadTVar` usage

fied any variable that T1 has read [12]. In case an application can have transactions with different readset sizes, the intuition then is that the bigger the readset, the more probabilitiy that the transaction will rollback. Figure 8 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 8 core executions since they have the greatest rollback rate. Only 9 out of the 13 applications in the suite are plotted because UnionFind and BlockWorld do not rollback in an 8 core configurations and SingleInt and TCache do not have transactions with variable readset length. As it could be seen, there is no value smaller than one, which means that the rolled back transactions have the same or a bigger readset size than the committed ones. Moreover, the plotted average having a value of 1.8 confirms the correlation between readset size and 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 and the TCache have a reads/writes ratio of 1, i.e. a read for each write performed. For the rest of the programs, this ratio is always greater than one, which supports the fact that most applications are read-dominated by nature.

4.4 Wasted and useful work and the breakdown of improvements

In a rollback situation, all of the operations perfomed by the transaction have to be ignored and re-executed; the time that this work took 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 10).

Following is an example showing the contribution of this work to the research community: It has been observed (Table 3, Figure 1 and [15]) that by utilizing `unreadTVar`, the execution of a program that traverses linked list gets much faster and more scalable. In that work, it is hypothesized that the causes of this important speedup can be the reduction of both: the wasted work and the commit phase duration. By looking at Figure 10 it is clear that `unreadTVar` helps to have much fewer rollbacks (and consequently less wasted work) so the first part of the hypothesis is confirmed. Moreover, Figure 9 shows that commit times are much lower

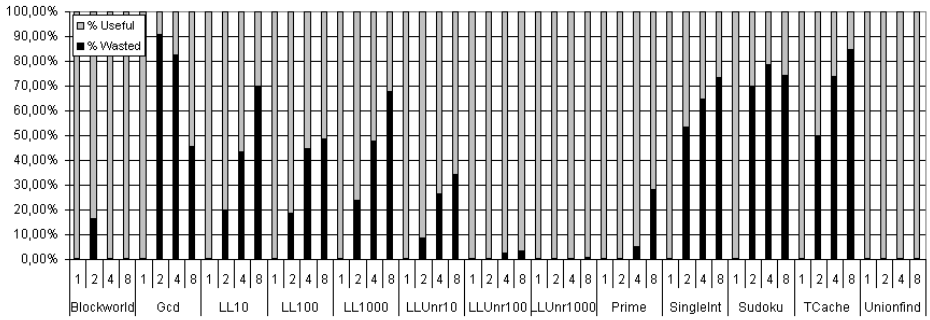

Figure 10. Percentage of wasted work per application

when `unreadTVar` is used, finally confirming all the predictions on the causes of the speedup.

The previous example shows how useful a profiling tool can be for researchers at the time of explaining results 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.

We believe that this metric is useful to indicate how well the applications scale transactionally: this could also be useful to decide the switchover point in mechanisms such as speculative lock-elision [19] in which the runtime switches from transactions to locks when there is too much wasted work.

5. Conclusions and Future Work

In this work, the internal behavior of several Haskell STM programs was analyzed. First, it is shown that one of the distinguishing factors of STM applications is the rollback rate (defined as number of rollbacks per committed transaction). The increase in the rollback rate as the number of cores is increased determines the transactional scalability of the application. On the same topic of rollbacks, we found that when the different threads update the same variable, there is no way of making the critical section parallel so it is intuitively expected to have a big number of rollbacks. Consequently, it can be also concluded based on the high number of times some transactions rollback (the authors coin the term "late commit" for this condition), that it might be necessary to tune the runtime system in order to avoid excessively aborting transactions. Finally on the topic of rollbacks, we experimentally verified our intuition that a bigger readset size is correlated with a greater probability of rollback.

Although providing the application with more cores might appear to be promising to increase performance, it was observed that average transactional time also increases in that situation. Particularly with a high number of cores, some transactions take too long to execute, therefore future research needs to further tackle the issue of whether the system architecture and thread scheduling conform to the demands of transactional management. In order to address this issue, we plan to run the benchmark on a 128 core SMP machine.

Another issue is the overheads associated with transactional management. 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 of the Glaskow Haskell Compiler, this is left as future work.

Another important contribution of this work is a first gathering together of Haskell STM applications that can serve as a benchmark suite. In the future, we are interested in further expanding this benchmark suite. In particular, we are especially interested in including networking applications such as TCP or HTTP servers.

Acknowledgements

The authors would like to thank Srdjan Stipic, Oriol Prat, Roberto Gioiosa, Edmund S. L. Lam and Martin Sulzmann for their useful suggestions and help.

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 TIN2004-07739-C02-01 and by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC).

References

[1] M. Herlihy and E. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures", in 20th Annual International Symposium on Computer Architecture, May 1993.

[2] N. Shavit and D. Touitou, "Software Transactional Memory", in Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing, pp. 204-213, 1995.

[3] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir and D. Nussbaum, "Hybrid Transactional Memory", in Proceedings of the Twelfth International Conference on

Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 2006.

[4] Man-Lap Li, Ruchira Sasanka, Sarita V. Adve, Yen-Kuang Chen and Eric Debes. "The alpbench benchmark suite for complex multimedia applications", in Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), 2005.

[5] K. Albayraktaroglu, A. Jaleel, X. Wu, M. Franklin, B. Jacob, C.-W. Tseng and D. Yeung, "BioBench: A benchmark suite of bioinformatics applications" in Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (IS-PASS 2005), March 2005.

[6] Ramanathan Narayanan, Berkin Ozisikyilmaz, Joseph Zambreno, Jayaprakash Pisharath, Gokhan Memik and Alok Choudhary. "MineBench: A Benchmark Suite for Data Mining Workloads", in Proceedings of the International Symposium on Workload Characterization (IISWC), October 2006.

[7] W. Partain. The nofib Benchmark Suite of Haskell Programs. Dept. of Computer Science, University of Glasgow, 1993.

[8] R. Guerraoui, M. Kapalka and J. Vitek, "STMBench7: A Bench-mark for Software Transactional Memory" in Proceedings of the Second European Systems Conference, March 2007.

[9] Haskell Official Site, http://www.haskell.org.

[10] Hal Daume III,"Yet Another Haskell Tutorial", www.cs.utah.edu/ hal/docs/daume02yaht.pdf

[11] S. Peyton-Jones, A. Gordon, and S. Finne, "Concurrent Haskell", ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (PoPL), 1996.

[12] T. Harris, S. Marlow, S. Peyton-Jones and M. Herlihy, "Composable Memory Transactions", in Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Chicago, IL, USA, June 15-17, 2005.

[13] Blundell, Colin and Lewis, E Christopher and Martin, Milo M. K., "Subtleties of Transactional Memory Atomicity Semantics", Computer Architecture Letters, Vol 5, Number 2, November 2006.

[14] Haskell and Web: Haskell Transactional Cache, http://haskell-web.blogspot.com/2006/11/transactional-cache-for-haskell.html

[15] N. Sonmez, C. Perfumo, S. Stipic, A. Cristal, O. S. Unsal and M. Valero, "UnreadTVar: Extending Haskell Software Transactional Memory for Performance", in Eighth Symposium on Trends in Functional Programming (TFP 2007), New York, April 2007.

[16] T. Skare and C. Kozyrakis, "Early Release: Friend or Foe?", Workshop on Transactional Memory Workloads, Ottawa, Canada, June 2006.

[17] E. S. L. Lam and M. Sulzmann. "A concurrent Constraint Handling Rules implementation in Haskell with software transactional memory" in Proc. of ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP'07), January 2007.

[18] T. Fruhwirth, "Parallelizing Union-Find in Constraint Handling Rules Using Confluence", 21st Conference on Logic Programming ICLP, October 2005.

[19] R. Rajwar and J.R. Goodman, "Speculative Lock Elision: Enabling Highly-Concurrent Multithreaded Execution", in Proceedings of the 34th International Symposium on Microarchitecture, 2001.