Profiling Transactional Memory applications on an atomic block basis: A Haskell case study

Nehir Sonmez[1,2] , Adrian Cristal[1], Tim Harris[3], Osman S. Unsal[1], and Mateo Valero[1,2]

[1] Barcelona Supercomputing Center
[2] Universitat Politècnica de Catalunya
[3] Microsoft Research Cambridge
{nehir.sonmez, adrian.cristal, osman.unsal, mateo.valero}@bsc.es, tharris@microsoft.com

Abstract. In many recent works in Transactional Memory (TM), researchers have profiled TM applications based on execution data using lumped per-transaction averages. This approach both omits meaningful profiling information that can be extracted from the transactional program and hides potentially useful clues that can be used to discover the bottlenecks of the TM system. In this study, we propose partitioning transactional programs into executions of their atomic blocks (AB), observing both the individual properties of these ABs, and their effects on the overall execution on a Software Transactional Memory (STM) benchmark in Haskell. Profiling on the AB-level and focusing on the intra-AB relationships helps to (i) characterize transactional programs with per-AB statistics, (ii) examine the conflict relationships that are caused between the ABs, and thus (iii) to identify and classify the shared data that often cause conflicts. Through experimentation, we show that the AB behavior in most of the Haskell STM benchmark applications is quite heterogeneous, proving the need for such fine-grained, per-AB profiling framework.

1 Introduction

Some 30 years ago, Lomet proposed an idea to support atomic operations in programming languages, similar to what had already existed in database systems [17]. More than a decade has passed since the first description of a Transactional Memory system in hardware (HTM) [13], and that of a software-only implementation [22]. Nowadays, TM is being seen as one of the most promising ways of the programming revolution which is almost late: many- and multi-cores are already dominant in the market. While the upcoming Rock processor [26] is proof that TM hardware is near, one thing is for sure: "Multicore architectures are an inflection point in mainstream software development because they force developers to write parallel programs" [1].

Software Transactional Memory (STM) promises to ease the difficulty of programming using conventional mechanisms such as locks in a threaded, concurrent environment. Locking has many problems: simple coarse-grained locking does not scale well, while more sophisticated fine-grained locking risks introducing deadlocks and data races. Many scalable libraries written using fine-grained locks cannot be easily composed in a way that retains scalability and avoids deadlock and data races. Although STM-based algorithms can be expected to run slower than ad-hoc non-blocking algorithms or fine-grained lock based code, they are as easy as using coarse-grained locks: one simply brackets the code that needs to be atomic.

Although the number of STM benchmarks in the literature keep increasing [3,9,18,19,21], very little experience exists on the characterization of TM programs.

Some relevant transactional attributes discussed in these benchmarks as well as in [5,6,29] include the total transactional time, the number of, and the time spent committing/aborting transactions, readset/writeset sizes and their corresponding reads/writes. However, while using lumped sum averages for characterization might be acceptable for some arbitrary execution, this is not the most useful way to profile TM programs. For instance, a rough estimate might be obtained by dividing the total number of transactional reads made by the program by the total number of transactions, to conclude that the program makes so many transactional reads per transaction, but we argue that such a statistic is mostly useless. Firstly, a program can be composed of various long-short (time), large-small (readset and writeset sizes) transactions and different abort rates, and an average value will omit many useful profiling data. Secondly, a typical program includes execution patterns, loops, function calls; deserving and perhaps requiring a better, finer-grained characterization framework. Furthermore, omitting such profiling information can result in ignoring simple pathological problems of the TM system.

Although an STM program runs transactions, these are actually executions of a specific atomic block, the piece of code inside the *atomically{}* , marked to run atomically to the rest of the system in an all-or-nothing fashion. In runtime, atomic blocks execute as transactions. To better reason about transactional attributes, we propose partitioning all benchmarks by their atomic blocks (AB), their bare transactional source code. In particular, the contributions of this paper are as follows:

- A Haskell benchmark suite is presented where each program is partitioned into its atomic blocks, and all TM attributes such as: the number of times committed/aborted, time spent doing work and time spent committing and aborting transactions, the number of reads and writes in committing and aborting transactions and readset and writeset sizes are grouped as executions of the particular atomic blocks. The AB behavior in most of the Haskell STM benchmark applications turns out to be quite heterogeneous, proving the need for finer-grained, per-AB profiling framework.
- Critical sections can be seen as AB conflict matrices of conflicting shared variables, which help to reason about the transactional aborts. This is achieved by collecting during runtime, per each AB, (i) the set of the conflicting shared data (in Haskell's terms, transactional variables, or *TVars*) (ii) other ABs that these *TVars* conflict on, and the number of times the exact scenario occurs.
- Due to the merits of profiling conflicting ABs, we show that some ABs tend to get scheduled to run concurrently and abort one another often, while others do not. Such findings point towards future per-AB runtime optimizations.

Up to date, to the best of our knowledge, a brief summary of per-AB statistics that include the number of commits, abort and readset sizes, has been featured only by the Intel STM Compiler [2]. The work in [27] can examine the AB conflicts in TM applications; however it does not feature any intra-AB dependence relationships or the identification of the shared data that cause transactions to conflict. One work that utilizes conflicting variables to make runtime decisions does not identify ABs [28]. Furthermore, [4, 8] are two recent works that recognize and optimize on some of the issues that can also be detected by our profiling mechanism.

On the next section, we give an introduction to the Haskell STM runtime environment, followed by a description of our modifications described on Section 2. Section 3 explains our scope in depth on a singly linked-list example. Section 4 presents the Haskell STM benchmark used and the corresponding results. Section 5 wraps up the conclusions.

TABLE I STM Operations in Haskell

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()`

2 Glasgow Haskell Compiler and Proposed Approach

2.1 The Glasgow Haskell Compiler and Runtime System

The Glasgow Haskell Compiler (GHC) [12] is a compilation and runtime environment for Haskell 98 [7], a pure, lazy, functional programming language. Since a few years now, the GHC has natively contained STM functions; abstractions for communicating between explicitly-forked threads using transactional variables, built into the Concurrent Haskell library [14]. STM can be expressed elegantly in such a declarative language, where Haskell's type system, particularly the monadic mechanism allows threads to access shared variables only when they are inside a transaction [10]. This restriction that guarantees strong atomicity can be violated under other programming paradigms, for example, as a result of access to memory locations through the use of pointers.

Haskell STM provides a safe way of accessing shared variables among concurrently running threads through the use of monads, allowing only I/O actions to be performed in the IO monad, and STM actions in the STM monad. This ensures that only STM actions and pure computation can be performed within a memory transaction, which also makes it possible to re-execute transactions.

Although the core of the language is very different from other languages such as C# or C++, the actual STM operations are used in a simple imperative style and the implementation uses the same techniques used in mainstream languages. Haskell has a small runtime system written in C, making it easy to experiment modifications. The STM support that has been present for some time led both researchers and functional programmers to write various applications, some of which were profiled in this work.

Threads in Haskell STM communicate by reading and writing transactional variables, or TVars, using a set of transactional operations, including allocating (*newTVar*), reading (*readTVar*), and writing (*writeTVar*) transactional variables (Table 1).

2.2 The transactional execution in GHC

In concurrent applications written in Haskell STM, an atomic block is constructed inside the *atomically{}* function. *Atomically{}* takes a memory transaction, of type *STM a*, and delivers an *I/O* action that, when performed, runs the transaction atomically with respect to all other memory transactions. This provides the sequence of code to be executed in an all-or-none fashion to the transactional management system. Following the transaction's start with *atomically{}*, transactional variables are created, read from and/or written to in program order, with the system maintaining a per-thread transaction log for all tentative accesses, called the transactional record (*TRec*). All the variables that are written to are called the "writeset" and all that are read from are called the "readset" of the transaction. These two sets usually overlap. This speculative execution, where the actual computational work takes place is called the work phase (WP).

Later comes the commit phase (CP), where the transaction might commit its changes into the memory. In this phase, where Haskell STM performs lazy conflict detection, the runtime system validates that the transaction was executed on a consistent system

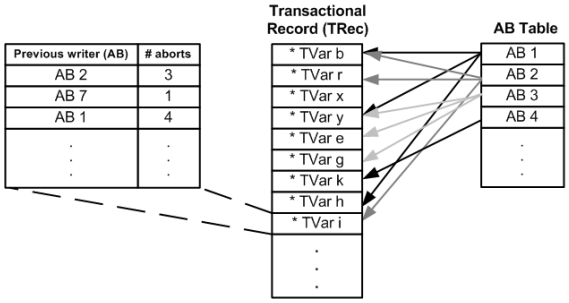

Fig. 1. Each AB keeps information on each conflicting TVar, its previous writer that caused the conflict, and the number of times each previously writing AB aborts the transaction on the specific TVar.

state, and that no other finished transaction may have modified relevant parts of this state in the meantime. First, a global lock is acquired (in the coarse-grained locking scheme of GHC). Next, validation is performed on the first part of the two-phase commit, going through the readset and the writeset, comparing each variable with its local original value that was obtained at the time of access. If all *TVars* accessed by the transaction turn out to be consistent, then all new values are actually committed into the memory in the second part of the two-phase commit. On encountering an inconsistent *TVar*, which means that some concurrent transaction committed conflicting updates, the system immediately aborts the transaction, discards the tentative modifications, and restarts the transaction. At the end of the CP, the global lock is released, which causes the course-grained locking technique to allow only one transaction to be in commit phase at a time. GHC also features a per-*TVar*, fine-grained locking version of the commit phase, however its behavior was not studied for this work, although the proposed profiling infrastructure would work with different locking implementations as well.

2.3 Conflicts and Aborts

A conflict occurs if two or more transactions access the same memory address and at least one of the accesses is a write. Transactions in Haskell STM that discover a conflict during validation immediately abort, discard all non-committed data, and restart. It might be useful to see all transactions in two groups: transactions that do not abort, and those that do. Inversely, transactions that make others abort and transactions that do not make abort exist, however these two categorizations are distinct. By grouping transactions as executions of ABs, it is possible to identify aborts in terms of specific conflicting *TVars* (*confTVar*) and see between which aborting and aborted (victim) ABs they occur. In the next section, we show how some *TVars* can be more involved in aborts than others. Having this information at runtime could make it possible to fashion optimization techniques that involve the aborting *TVars*.

The biggest overheads in STM come from keeping track of transactional variables, and causing wasted work by aborting the transactions in case of a conflict. Previous work in [19] shows wasted work for up to 50% on the Haskell benchmark. Some of those programs are also used on this work. Aborts can be costly depending on their frequency, and the amount of work done inside the transaction. Repetitive aborts degrade the performance of the program. One motivation for this work is to find out more about how aborts happen and if similar or repetitive aborts exist.

2.4 GHC Hooks

Precisely what makes AB-based profiling possible is that each atomic block execution, being a specific function call, is pointed to in GHC by a variable called the *atomically_frame_pointer*. This provides a unique identifier in runtime to group

atomic blocks encountered in the code, pointing to the beginning of the code inside of *atomically{}*. Although it would require modifications to the program code, another way of uniquely identifying ABs is by passing an explicit additional parameter to the atomically function, such as an integer constant, and then partitioning the profiling information accordingly. Such an approach of finding the appropriate AB identifier and accordingly grouping transactional profiling data should be feasible under many other STM systems.

ABs are added to a global AB Table (Fig. 1) when *atomically{}* is invoked. During CP, at the time of validation where the contents of the *TVars* are checked for consistency with their prior-to-start transactional values, once a *TVar* fails, the AB Table is updated with this new occurrence of a conflict along with (1) the AB that is currently being aborted, (2) the specific conflicting *TVar*, (3) the AB that last wrote to that *TVar*, causing the abort, and (4) the number of times this exact scenario occurred.

3 An Example: Profiling a Linked-List

To demonstrate the proposed approach, a transactional linked-list (LL) was implemented in Haskell performing 10% deletes, 10% inserts and 80% lookups. The links between the nodes (Fig. 2) are represented as *TVars*, therefore during list traversal, *TVars* are accumulated into the transaction's readset. Although this is not the most suitable concurrent implementation of a linked structure [24], it's a simple example that should serve to demonstrate our approach. Throughout the program, the atomic blocks are commented with "ABx" denotations.

The *main* function in line 36 in Fig. 3 takes the necessary arguments from the user (the number of operations, the list capacity, and the number of threads to fork) and calls *main1*. Although Haskell is a language with lazy evaluation, the *do{}* "syntactic sugar" enables to construct of a sequence of actions. Line 22 in *main1* is the first atomic block, AB1, where the initial half-list is created atomically, and following that, AB2 outputs the number of elements that it contains. Later, the Haskell synchronization variables (*MVars*) are created. Then, a desired number of threads are forked, each performing a lookup/insertion/deletion of a random integer, executing AB3, AB4 or AB5 respectively; and (ideally) performing deletes a tenth of the time, inserts another tenth, and on the rest doing lookups. When the total number of operations complete running on the threads, the *MVars* are taken: this barrier will make sure that all threads finish executing before the main thread ends. By invoking AB6 and AB7 on lines 29 and 30, the program will print the ultimate number of elements that the list contains. Finally, before the end, *ReadTStats*, a function that enables printing out the gathered statistics is invoked. In this program, AB1-AB2, and AB6-AB7 pairs can serve to illustrate the similarities, and AB3-AB4-AB5 the variation between atomic blocks. Due to space limitations, we cannot provide all the used functions, however a similar insert function can be found in [24]. The insert and delete functions (AB4 and AB5 respectively) are similar, except that delete performs one more *readTVar* because it needs to read two nodes ahead.

```
data LinkedList =
     Start  {nextNode :: TVar LinkedList}
     | Node {val :: Int, nextN :: TVar LinkedList}
     | Nil
```

Fig. 2. The data declaration of the transactional linked-list.

```
createThread :: Int -> TVar ListNode -> Int -> MVar Int -> IO ThreadId
createThread numOps tList maxNumber mvar =
     forkIO ( do { callNTimes numOps
                   (do
                   { rnd1 <- randomRIO (1::Int, 10)
                   ; rnd2 <- randomRIO (1::Int, maxNumber)
                   ; case rnd1 of
                   ; 1 -> do { atomically (deleteListNode tList rnd2) -- AB5
                             ; return ()}
                   ; 2 -> do { atomically (insertListNode tList rnd2) -- AB4
                             ; return ()}
                   ; otherwise -> do { atomically (lookupListNode tList rnd2) -- AB3
                             ; return ()}
                   })
                   ; putMVar mvar 1
                 }  )

createThreads :: Int -> Int -> TVar ListNode -> Int -> [MVar Int] -> IO()
createThreads n numOps tList maxNumber mvars
            = mapM_ (createThread numOps tList maxNumber) mvars

main1 :: Int -> Int -> Int -> IO ()
main1 numops listLength numThreads = do
     { ourList <- atomically (createSampleList (reverse[x|x<-[1..listLength],((mod x 2)==0)])) --AB1
     ; ourListAsString <- atomically (toString ourList) --AB2
     ; putStrLn (show (length ourListAsString))
     ; ourTList <- newTVarIO ourList
     ; mvars <- replicateM numThreads newEmptyMVar
     ; threads <- createThreads numThreads numops ourTList listLength mvars
     ; mapM_ takeMVar mvars
     ; ourList2 <- atomically(readTVar ourTList) --AB6
     ; listAsString <- atomically (toString ourList2) --AB7
     ; putStrLn (show (length listAsString))
     ; stats <- readTStats
     ; putStrLn (show stats)
     ; return ()
     }

main :: IO()
main = do { args <- getArgs
     ; let numops = read (args!!0)
     ; let listlength = read (args!!1)
     ; let numthreads = read (args!!2)
     ; main1 numops listlength numthreads}
```

Fig. 3. Major functions in the linked list code

3.1 Observed Statistics

All executions in this work, including the LL example were run on a 128-core SGI Altix 4700 system with 64 dual-core Montecito (IA-64) CPUs running at 1,6 GHz. The command "*MainLL 16000 100 8 +RTS –N8*" was executed to run the program on Fig. 3: 16000 is the total number of operations, 100, the maximum list size, and 8 the number of Haskell threads to use. The RTS option *–Nx* lets the user choose the number of OS threads to utilize, and since there are enough cores in the system, there is one thread per core.

The seven atomic blocks of the linked-list program and their individual statistics can

TABLE II AB Properties of the Linked List program

AB	# A	# C	write set	Read-only set	Total writes (A)	Total reads (A)	Total writes (C)	Total reads (C)	CP (A) ‡	CP+WP (A) ‡	Val (C) ‡	CP (C) ‡	CP+WP (C) ‡	Total Conf TVar
AB1	0	1	1	1	0	0	50	100	0	0	4,732	10,290	967,093	0
AB2	0	1	0	51	0	0	0	51	0	0	4,105	8,393	168,660	0
AB3	2,735	12,745	0	348,072	0	207,283	0	670,654	1,116,122	2,558,596	1,097,159	1,211,440	5,206,129	542
AB4	187	1,690	807	45,280	81	13,799	807	88,794	61,534	107,894	146,314	153,328	379,105	129
AB5	213	1,565	800	42,638	107	23,226	800	120,950	67,174	157,486	132,227	139,859	401,216	147
AB6	0	1	0	1	0	0	0	1	0	0	3,135	8,974	48,316	0
AB7	0	1	0	58	0	0	0	58	0	0	4,296	8,425	292,619	0
total	**3,135**	**16,004**	**1,608**	**436,101**	**188**	**244,308**	**1,657**	**880,608**	**1,244,830**	**2,823,976**	**1,391,968**	**1,540,709**	**7,463,138**	**560**

‡: in 1000s of cycles, A: Aborted transactions, C: Committed transactions, Val: Validation, CP: Commit Phase, WP: Work Phase

be seen on Table 2. There are three ABs that get aborted, and four that do not—basically in this example, the heavyweight atomic blocks AB3, AB4 and AB5 tend to get scheduled concurrently and cause conflicts.

The total transactional execution distribution can be seen in Figure 4. The wasted work caused by aborting transactions constitutes 14% of the overall transactional runtime. 78% of this is due to aborts of the atomic block AB3, which in total takes up 82% of the transactional execution, either committing or aborting. AB3 (lookup) commit counts are an expected eight times to those of the atomic blocks AB4 and AB5. Its writeset, commit and abort writes are zero, since it only performs lookup. Comparing AB1 and AB6, two atomic blocks with single element readsets, AB1 actually creates a list of 50 elements (writes:50), traversing all the possible nodes to create all the even numbers from 1 to 100 (reads: 100). AB6 does less work, only reading the ultimate set, which turns out to contain fifty-seven (plus one for the start node) elements in the end. This can be seen as well on the transactional reads and the readset statistics of AB7, which is very similar again to those of AB2.

For committed transactions, we profile executions in three parts: The work phase, the validation inside the commit phase, and the rest of the CP, writeback, where the writes are actually committed to memory. GHC will still check all *TVars* to see if any need to be written back, this is the reason why the atomic block that corresponds to the lookups, AB3, also spends time in the commit phase in aborted transactions. Read-only atomic blocks like AB3 cannot be aborters of other ABs.

3.2 Conflicting TVars and Aborts

The final column on Table 2, *confTVar*, indicates on how many distinct TVars the aborted transactions are due to. Our profiling mechanism collects information about all aborts during runtime where these conflicting TVars as well as the corresponding aborting/victim transactions can be observed. Theoretically, each run could have a different set of *TVars*, depending for example, on the scheduling order of the atomic blocks, the loops in the code, the randomized variables etc, so ABs do not have to exhibit the same behavior on each execution. However, for our benchmark, we generally see quite stable and meaningful behavior.

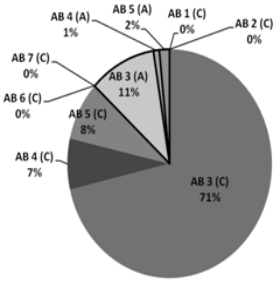

Fig. 4. Distribution of the total transactional execution (C: Committed transactions, A: Aborted transactions)

TABLE III
THE DISTRIBUTION OF CONFLICTING TVARS

Aborter/ victim	AB3	AB4	AB5	*total*
AB3	0	0	0	
AB4	362	82	77	52
AB5	356	69	87	51
total	**718**	**151**	**164**	**103**
confTVars	542	129	147	81
difference	**176**	**22**	**17**	**21**

TABLE IV
THE DISTRIBUTION OF ABORTS VS CONFLICTING TVARS

Aborter/ victim	AB3	AB4	AB5	*total*	AB3	AB4	AB5
AB3	0%	0%	0%	0%	0	0	0
AB4	44%	3%	3%	50%	65%	15%	14%
AB5	43%	3%	4%	50%	64%	12%	16%
total	87%	6%	7%				

TABLE V
THE DIFFERENCE: AB ABORT DUO

Aborter/ victim	AB3	AB4	AB
AB3	0	0	
AB4	122	14	1
AB5	54	8	
total	176	22	1

Table 3 shows the distribution of the *confTVars* that were collected during the execution in terms of the aborter and the victim ABs. *ConfTVars* is the number of distinct *TVars* inside the AB that caused an abort at least once. However, they can appear in more than one AB, therefore the total number of conflicting *TVars* on Table 2 is not the sum of all *confTVars* in all ABs: this sum instead, is the total number of unique *TVars* that conflicted. On the left side in Table 4, all 3,135 aborts that occurred are classified into aborter and victim AB groups, and on the right side, the corresponding shared conflicting *TVars* are depicted. For example, the case where AB5 caused an abort on AB3 constitutes 43% of the total aborts, but involves 64% of the total critical section (the conflicting *TVars*). Inversely, the case of two concurrent AB5 executions aborting one another occurs in 4% of the total aborts but involves 16% of the conflicting *TVars*. Overall, AB3 was a victim (was aborted) in 87% of the total aborts. Please note that the aborter/victim matrix can be highly asymmetric; this heterogeneous behavior is one of the key reasons why we argue for profiling on an AB granularity.

The shared *confTVars* (the row "difference" on Table 3) gives way to Table 5. For example, the conflicting *TVars* that caused an execution of AB4 to abort another concurrent AB4 execution, also caused aborts with AB5 in 36% of the total aborts of AB4 (8/(14+8)). Table 5 highlights those "hot" *TVars* that cause aborts in many ABs. Such information might be useful for resolving conflicts: Firstly, it might be a good idea to omit the scheduling of an AB concurrently with itself, if it is known to abort with itself. Furthermore, ABs forming such "conflict duos" (and triples, etc) could be scheduled with more care. Such an approach could diminish aborts that harm performance. The studies in [4,8,23,28], as well as work on contention management [20] act upon similar observations.

With a simple program such as the LL, we can see how all statistics are partitioned in different ABs. In the next section, we look at some other characteristics of ABs using a Haskell STM benchmark.

3.3 The Overhead

The atomic blocks give us a possibility that didn't exist before: to group, compare and view the interactions of transactional executions. However, in this endeavor, our approach needs to introduce new fields to the *TVar* structures such as the ones depicted on Fig. 1, as well as others that are used to accumulate the necessary information for transactional statistics, since GHC recycles the *TRec* structures that it uses for each transaction. New fields for all statistics were added to the *TRec* to be used for keeping related information on the number of aborted/committed transactions, the readset/writesets, number of transactional reads/writes and Validation/CP/WP runtimes in cycles.

These additions do add some overhead, but it is the most straight-forward way to achieve our objective. The slowdown introduced by the profiling mechanism depends on the number of ABs and the size of the *TVar* set that is saved for profiling. Although looking into a range of 5 ABs and hundred *confTVars* works faster than 20 ABs with a thousand conflicting *TVars* each, it still causes on average 7% more aborts on the system running the linked list application. Although the impact on aborts is small, more work is being carried on to reduce this overhead.

4 The Extended Haskell STM Benchmark

The benchmark consists of several programs from the Haskell benchmark in [19], alongside a Hash Table implementation [15] and a Parallel Sudoku Solver [25]. Some of the programs in this benchmark spend as much time aborting transactions as

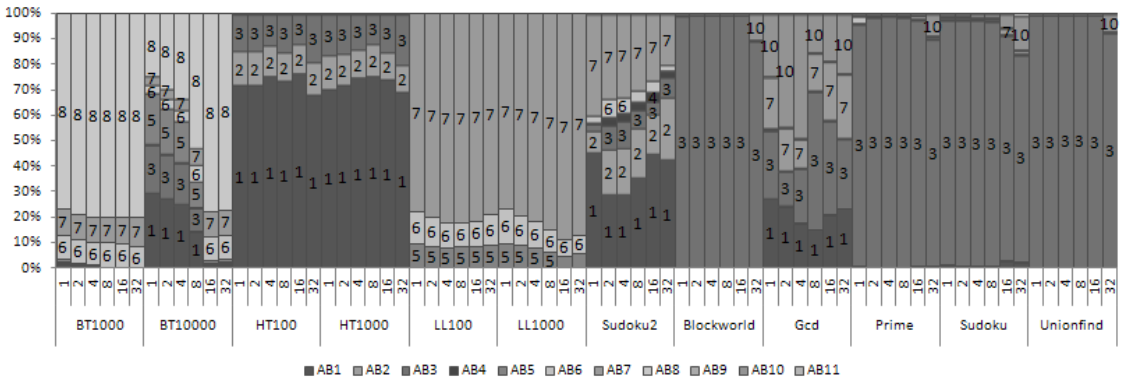

Fig. 5. Overall distribution of transactional time of committed transactions per atomic block on all benchmarks.

committing them. All of the reported results are based on the average of five executions on the Altix 4700 system. Up to 32 cores were used for the clarity of graphs. The outputs were sorted by the atomic block identifier, and then named AB1 through ABx, however they do not represent any initial order of execution. Please note that the atomic blocks with the same numbers are consistent in different runs of the same program but distinct among the benchmarks.

4.1 Transactional occupation per AB on committed transactions

Figure 5 shows the distribution of ABs on the whole benchmark based on the total transactional time spent on committed transactions, which represents the total amount of useful work that each AB had to do for each program to complete. Clearly, we can see both heterogeneous (a variation among the atomic blocks of BT10000, Sudoku2, GCD), and homogeneous behavior (the Hash Table and the Linked List are always performing similarly in all core counts). The heterogeneity exists inside the programs that make use of the CCHR compiler as well: GCD runs differently from the other applications of the CCHR suite.

The predominant AB in BT/LL/HT benchmarks is always the lookup function, however, the initial list creation that creates a tree of 5,000 elements takes up a significant amount of time in the BT 10000 implementation, especially in 1-8 core runs. AB8 of the BT is the lookup function, which expectedly runs eight times as long as AB6 and AB7, (which insert and delete) however, the atomic blocks that do list creation and print also spend a significant amount of time in the WP, independent of the number of cores.

4.2 Abort Analysis per AB

Figure 6 shows the ratio of aborts to committed transactions, only for the atomic blocks that get aborted. Although the BT, HT and the LL are similar programs with each having heavyweight ABs for insert, delete and lookup functions, they have different abort rates. The HT and the LL are heavy aborters especially with high core counts. However, the BT, being a much more concurrent application aborts very little, especially with larger trees.

Most of the ABs demonstrated in CCHR applications that abort also abort more in larger core counts with varying rates per program. However, AB5 (pictured separately) aborts fourfold to commits. In reality, AB5 is a small atomic block with a readset that is equal to the number of forked threads, and a writeset of zero, that only executes as a transaction once, but causes up to four aborts running on 32 cores, or commits on the first try on lower core counts. Sudoku2 also has two aborting ABs

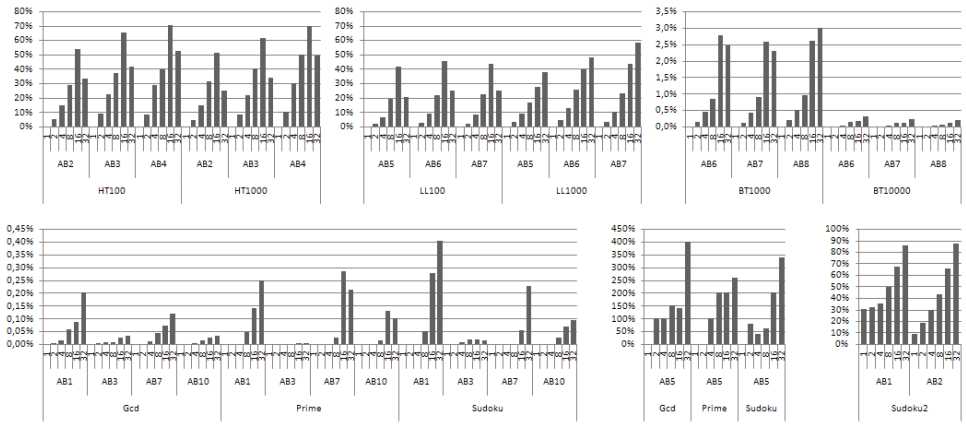

Fig. 6. Percentage of aborts per commits, on aborting ABs of all programs.

that abort quite frequently with more available concurrency, where the rest of the atomic blocks always succeed in committing.

Pushing concurrency on a not-so-parallel application, serialized commits (coarse-grained locking) and the serial, stop-all-the-world garbage collection all hurt the scalability of the program [19]. In other cases, the cache performance and other variables can easily affect the overall system performance.

4.3 Per-AB commit phase and validation

Figure 7 (last page) shows how each AB spends time during executions that end up in commits, in work phase and in commit phase (validation and writeback). Although they all total a 100%, the graphs had to be pruned for clarity, and only selected executions are depicted. The Sudoku2 application contains quite heterogeneous ABs in terms of validation and overall commit phase behavior. Almost all ABs scale poorly: although some writeback phases at first seem to scale better with more cores, the time spent on validation is quite overwhelming.

The LL spends less time in CP using 1000 elements, since the WP is now larger, operating on a bigger list. Although for the CP more work has to be done as well, the

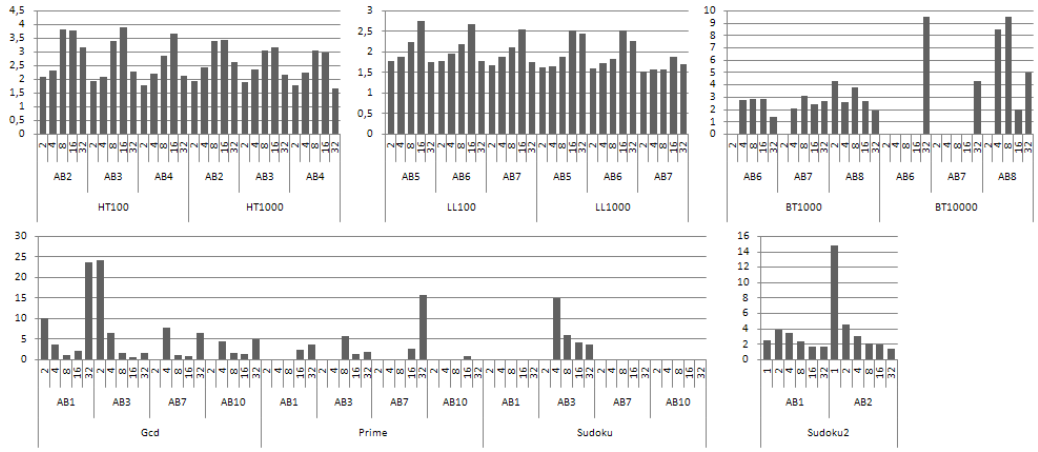

Fig. 8. The ratio of the per-transaction average runtime of aborted transactions to per transaction average runtime of committed transactions. It can be seen that transactions that end up aborting in the end take up a lot more runtime that those that manage to commit.

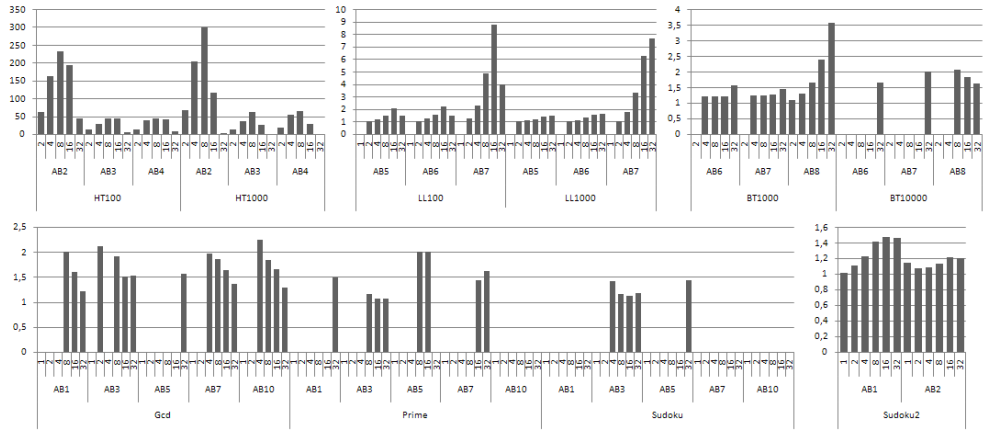

Fig. 9. Average number of aborts per each conflicting TVar.

insert/delete/lookup operations themselves take up more execution time. The same is also true for AB7 which is the lookup atomic block, where the CP seems to be less, while it is the WP that is getting huge. On the HT, the lookup function is AB2, where the CP again dominates an important part of the transactional execution.

The last subfigure in Figure 7 belongs to the programs that utilize the CCHR compiler [16], where same ABs can function differently solving different programs. Regardless of knowing the functionality of ABs inside such a compiler or a library, one can profile the behavior and attempt to optimize it accordingly. For example, AB5 has a CP that takes less percentage of time with more cores, which might be desirable, but a WP that always increases. AB9 does the opposite and AB4 seems to have an almost fixed distribution among all cores. To identify TM performance bottlenecks, besides knowing the per-AB runtimes, it might be useful to see on which phase each AB is the busiest inside the transactions.

4.4 Per-transaction Runtime Averages of Aborted and Committed Transactions

As mentioned on Section 2.B, a transaction in Haskell aborts as soon as an inconsistent *TVar* is encountered. Normally one might be tempted to think that aborted transactions would spend less time than committed transactions: The same amount of time in the work phase, and less time in validation (since abort's validation fails), meanwhile writeback is not performed for aborted transactions. However, Figure 8 shows that this is not true at all, and since commits are serialized, transactions wait on the coarse-grained lock. Those that manage to commit could also wait on locks, but it can be seen here that aborted transactions take a longer time waiting and those transactions that wait too long tend to abort in the end with a higher probability.

4.5 Aborts per Conflicting TVars

Aborts per *confTVars* (Fig. 9) is a "congestion metric" that shows how many aborts each conflicting *TVar* causes on average inside a specific AB. This is due to the frequency of repetitive aborts, because conflicted *TVars* are shared with concurrently scheduled instances of the some AB, or with other ABs. The larger the ratio of these two values (notice that by definition it has to be larger than 1), the more congested the AB is. For example, for a highly non-concurrent *singleInt* benchmark [19], this ratio would be huge, with one conflicting *TVar* but a very high number of aborts due to that *TVar*.

The Hash Table has a small and a very congested critical region. The Linked List is also similar, especially since the nodes close to the beginning of the list are very frequently accessed. The BT is somewhat more concurrently accessible than the LL.

Sudoku2 has two ABs out of eight that abort, but on average the lowest aborts per conflicting TVar ratio in the benchmark. Although its abort ratio (Fig. 6) is one of the highest, it can be concluded that its critical region is also large and it contains many *TVars* that can cause conflicts. This metric is important since such conflicting *TVars* can introduce many aborts, causing performance degradation and scalability problems in the system, and it might be useful to treat them accordingly, as done in [23] and [28].

5 Conclusions

Per-atomic block profiling can be a useful tool for many reasons. For simple micro benchmarks, it helps to see under the hood. For others, when it is not possible or doesn't make sense to know the behavior of each AB, this approach helps to recognize the transactional execution better and to identify certain AB behaviors such as identifying read-only and write-only, or heavyweight ABs, and isolating AB-related problems such as self-conflicting ABs or groups of ABs that abort each other and to construct conflict matrices. Clustering aborting TVars inside atomic blocks also provides an opportunity to identify conflicting shared data sets and to reason about the aborts that take place. In the future it might also be interesting to identify ABs that include nesting or IO in order to try to make appropriate runtime decisions.
We argue for and show through profiling TM applications, why a finer-granularity AB-based approach has to replace lumped averages. Our approach is general enough to be used with different atomic block implementations, eager conflict detection/resolution, or with other STMs.

Acknowledgements: This work is supported by the cooperation agreement between the Barcelona Supercomputing Center - National Supercomputer Facility and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625, by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC) and by the European Commission FP7 project VELOX (216852).

References

[1] A. Adl-Tabatabai, C. Kozyrakis and B. Saha, Unlocking concurrency, ACM Queue, vol. 4/10, pp 24-33, NY, USA, 2007.

[2] A. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha and T. Shpeisman, "Compiler and runtime support for efficient software transactional memory" in PLDI 2006, pp. 26-37, Ottawa, Canada.

[3] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, and I. Watson, "Lee-TM: A Non-trivial Benchmark for Transactional Memory", in Proc. ICA3PP 2008, Cyprus, June 2008.

[4] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. Kirkham and I. Watson, "Steal-on-abort: Dynamic Transaction Reordering to Reduce Conflicts in Transactional Memory", in SHCMP'08, June 2008.

[5] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas. "Bulk Disambiguation of Speculative Threads in Multiprocessors", in International Symposium on Computer Architecture (ISCA 2006), June 2006.

[6] 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 Proc. 12th HPCA. Feb 2006.
[7] H. Daume III. Yet another Haskell tutorial. In www.cs.utah.edu/ hal/docs/daume02yaht.pdf. 2002-2006.
[8] S. Dolev, D. Hendler, and A. Suissa, "CAR-STM: Scheduling-based collision avoidance and resolution for Software Transactional Memory", in PODC '08: Proceedings of the twenty-seventh ACM symposium on Principles of Distributed Computing, pp. 125–134, New York, NY, USA, 2008.
[9] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: A benchmark for software transactional memory. In Proc. of EuroSys2007, 315–324. ACM, Mar 2007.
[10] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In PPoPP'05, Chicago, USA, June 2005.
[11] T. Harris and S. Peyton-Jones. Transactional memory with data invariants. In First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, 2006.
[12] Haskell official site: http://www.haskell.org.
[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 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp 295–308, Florida, 1996.
[15] E. Kmett, An STM-based hash, http://comonad.com/haskell/thash/.
[16] E. S. L. Lam and M. Sulzmann. A concurrent constraint handling rules implementation in Haskell with software transactional memory. In DAMP'07, Jan 2007.
[17] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In Proceedings of an ACM conference on Language design for reliable software, pp 128-137, 1977.
[18] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing", in proceedings of IISWC 2008, September 2008.
[19] C. Perfumo, N. Sonmez, S. Stipic, O. Unsal, A. Cristal, T. Harris and M. Valero, "The limits of Software Transactional Memory (STM): Dissecting Haskell STM applications on a many-core environment", in Proceedings of the 2008 conference on Computing Frontiers, pp. 67–78, New York, NY, USA, 2008.
[20] W. N. Scherer and M. L. Scott, Advanced contention management for dynamic software transactional memory, in Proceedings of PODC '05, pp. 240-248, Las Vegas, USA, 2005.
[21] L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. "Delaunay triangulation with transactions and barriers", in IISWC 2007, pp. 107-113, 2007.
[22] N. Shavit and D. Touitou. Software transactional memory. In Symposium on Principles of Distributed Computing, pages 204–213, 1995.
[23] N. Sonmez, A. Cristal, T. Harris, O. S. Unsal, M. Valero, "Taking the heat off transactions: dynamic selection of pessimistic concurrency control", to appear in IPDPS 2009, Rome, Italy, May 2009.
[24] N. Sonmez, C. Perfumo, S. Stipic, A. Cristal, O. S. Unsal, and M. Valero, unreadTVar: Extending Haskell software transactional memory for performance. In Trends in Functional Programming Volume 8, Chapter 6, pp. 89-114, NY, USA, Apr 2007.
[25] W. Swierstra, A parallel version of Richard Bird's function pearl solver, http://www.haskell.org/haskellwiki/Sudoku#A_parallel_solver.
[26] M. Tremblay and S. Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread CMT SPARC processor. In IEEE International Solid-State Circuits Conference, Feb. 2008.
[27] C. von Praun, R. Bordawekar, and C. Cascaval, "Modeling Optimistic Concurrency using Quantitative Dependence Analysis", Symposium on Principles and Practice of Parallel Programming (PPoPP 2008), pp.185-196, UT, USA, February 2008.
[28] M. M. Waliullah and P. Stenstrom, "Intermediate Checkpointing with Conflicting Access Prediction in Transactional Memory Systems", In Proceedings of the 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS), Miami, Florida USA, Apr 2008.

[29] L. Yen, J. Bobba, M. M. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "LogTM-SE: Decoupling hardware transactional memory from caches".In Proc. HPCA, pp. 261-272, Feb 2007.

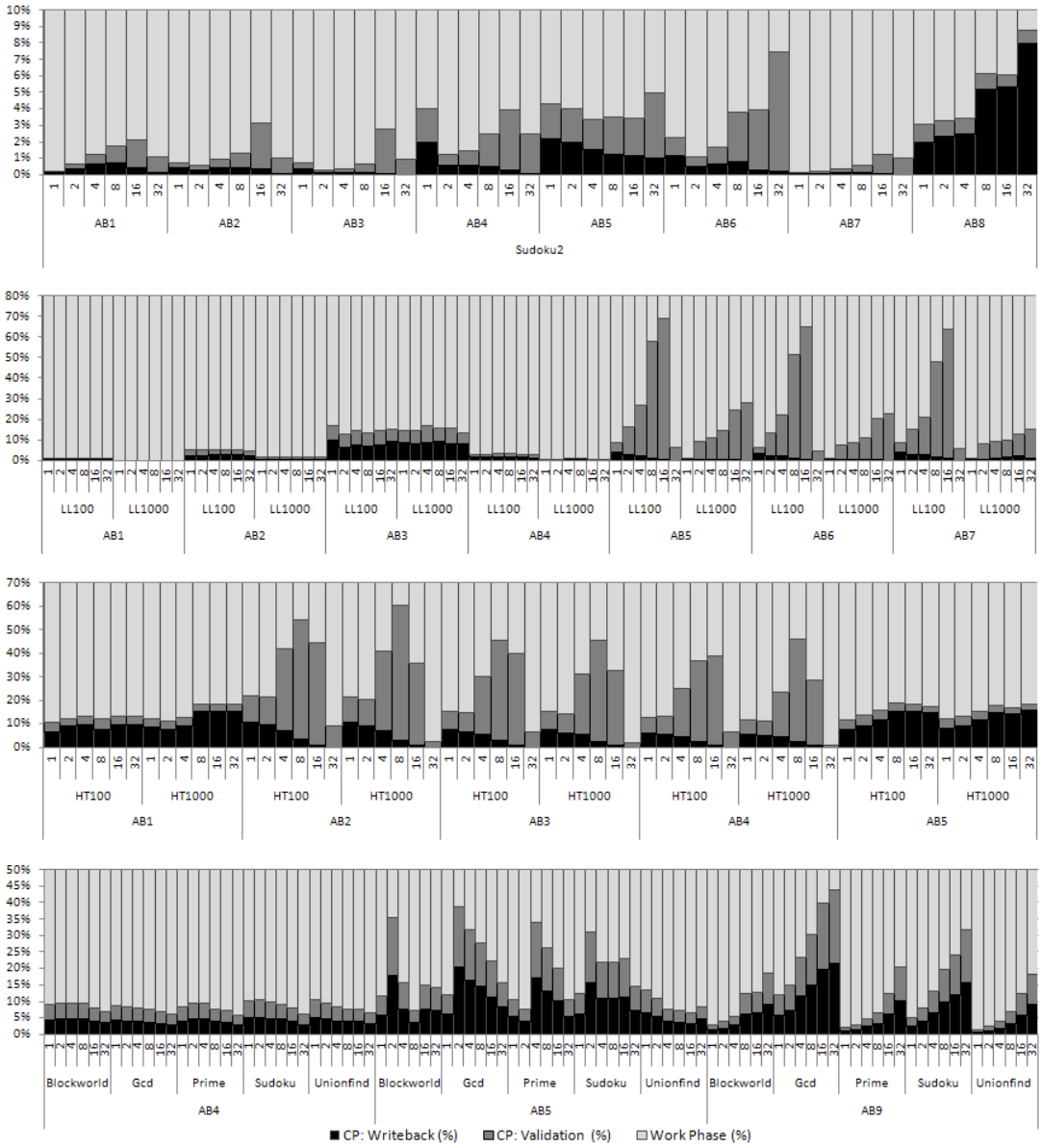

Fig. 7. Percentage of validation, writeback and the work phase on committed transactions.