Log based Transactions for Single-threaded Programs

Arini Balakrishnan
arinib@cs.wisc.edu

Piramanayagam Arumuga Nainar
arumuga@cs.wisc.edu

Abstract

Transactions are used for concurrency control in parallel programs. They allow programmers to identify sections of code as atomic, leaving the implementation to provide atomicity and isolation. Atomic blocks are also used for exception recovery, where the updates of a transaction can be rolled back in case of an exception. In this project, we develop a log-based transaction system based on LogTM, a hardware transactional memory. Our system guarantees atomicity of transactions (and not isolation). It provides the three transaction primitives, begin, commit and abort. It also supports nested transactions. Evaluation of the system on some selected applications shows an average overhead of 50 cycles per store and 10 cycles per abort.

1 Introduction

A transaction is a block of statements that is guaranteed to execute atomically and in isolation from other transactions. Some machine instructions like *test-and-set* and *swap* perform a predefined operation atomically and in isolation. Transactions can be viewed as a generalization of this idea, where the programmer can specify the operation to be executed atomically. Transactions are widely used in database systems for concurrency control. [4] proposes an architecture called Transactional Memory to provide synchronization in parallel programs. Transactional Memory provides a simple and intuitive interface to write parallel programs. It avoids common problems in lock-based techniques like priority inversion, convoying and deadlocks. Transactions can also be used in exception recovery. Exception recovery involves restoring program state and invariants and freeing resources when an exception occurs. This can be hard to implement and difficult to test [9]. The atomicity guaranteed by transactions can simplify restoring program state and invariants. For example, consider a function that moves an element from one list to another. Suppose, an exception occurs after the removal is performed, but before the insertion completes. In such a situation, the lists can be restored

to their initial state by performing an abort operation. On the other hand, error recovery in a non-transactional world is considerably complicated.

In this project, we implement a software transaction system for SPARC processors based on LogTM [7], a hardware transactional memory. It is implemented using EEL [6], an Executable Editing Library for SPARC binaries. Our system guarantees atomicity but doesnot provide isolation. So, it is applicable only to single-threaded programs. It supports the three transaction operations: *begin*, *commit* and *abort*. The $begin$ primitive marks the beginning of a transaction. $Commit$ marks the completion of the transaction. $Abort$ is used in case of an exception, to undo and discard the work done by the transaction. Our implemetation also supports nested transactions. We use data-flow analysis to identify program points that occur within transactions. We evaluate the system on two encryption algorithms (MD5 and SHA), quicksort and matrix multiplication. They showed an average overhead of 50 cycles per store and 10 cycles per abort. Overall execution overhead varied from 36% to 310%.

Section 2 discusses some background information related to our project. Section 2.1 discusses the motivation behind Transactional Memories and Section 2.2 introduces a particular implementation namely LogTM. Section 2.3 discusses about EEL, a library which we use to implement our project. Section 3 gives a high level design of our project. Section 4 discusses the implementation details and the results are presented in Section 5. We discuss some directions for future work in section 6 and conclude in section 7.

2 Background

2.1 Transactional Memory

Programmers need to adhere to strict programming discipline while writing multithreaded programs. When a location is shared between threads, accesses to it must be controlled by a lock. Fine grained locks provide concurrency, but are more complex and often lead to deadlock situations. Coarse grained locks, on the other hand do not scale well. Thus there is a tradeoff between concurrency and ease of programming. Also, lock based synchronization suffers from problems such as deadlocks, priority inversion, convoying and lack of fault tolerance. Transactional memory is an optimistic concurrency technique proposed as an alternative to pessimistic lock based scheme. Transactions provide optimistic concurrency control by allowing multiple transactions to execute concurrently. In case of a conflict between two transactions, one of the transactions proceeds to completion and the other gets aborted. The programmer need not concentrate on concurrency as

transactional memory handles it implicitly. Transactional memories could be implemented either in hardware ([7] [2] [1]) or in software ([8]).

2.2 LogTM: Log based Transactional Memory

LogTM is an implementation of transactional memory that performs eager version management. It makes the common case faster by storing new values in place and old values in a log. It handles commits and logging in hardware and aborts in software. LogTM handles $commits$ by discarding the log and resetting a log pointer. During an $abort$, it walks the log and restores the values. We implement a similar log-based transactional system in software. While LogTM performs logging at the granularity of the cache block, we do it at the granularity of a word.

2.3 EEL: Executable Editing Library

EEL (Executable Editing Library) is a library for analyzing and modifying an executable (compiled) program. It provides 4 abstractions: *executable, routine, control-flow graph (CFG) and instruction* for analyzing the executables at different granularities. It provides a *snippet* abstraction to encapsulate new code that is added to an executable. It also supports interprocedural analysis by providing an abstraction for the call-graph of the executable.

3 Design

Figure 1 gives a high level view of the design of the system. Our system has two components, the header file *transact.h* and the *executable editor*. The header file has the definitions for the transaction primitives, *begin, commit* and *abort*. These definitions are inline SPARC assembly instructions that copy a value into the register %g0. These instructions are essentially nops because %g0 is immutable. The value of the first operand in the instruction is used to distinguish between the different transaction primitives. The executable editor identifies the instructions corresponding to *begin, commit* and $abort$ and replaces them with code snippets that implement the semantics of these functions. The executable editor also inserts snippets that log old values before each store inside a transaction.

The system uses the following three structures to implement the semantics of a transaction.

- **Log:** It is used to store the old value at the memory locations updated during a transaction. We maintain

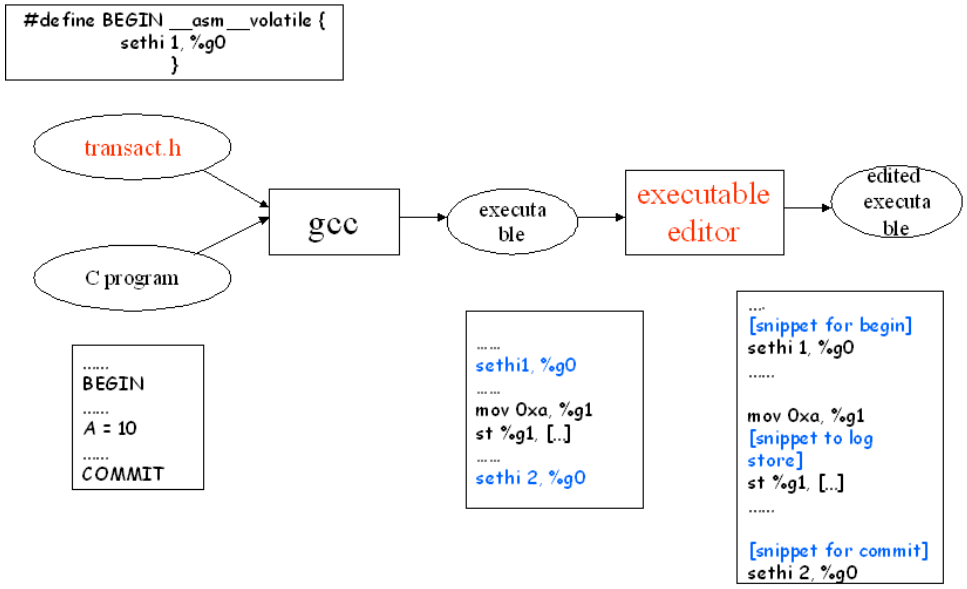


Figure 1: **System Design.** *The high level design of the system.*

a pointer to the next empty location in the log called the log-pointer (log-ptr).

- **Begin-pointer Stack:** The Begin-pointer Stack (BS) is introduced to support nested transaction. Nested transactions are discussed in section 3.1.

- **Register Checkpoint Stack:** During an $abort$, register modifications done by the transaction must be undone. Since the register file is considerably small we checkpoint the registers before each $begin$ and replay values from the register checkpoint stack during an $abort$. The Register Checkpoint Stack (RCS) is used for this purpose. To avoid checkpointing the entire register window, we disable register windows by giving the $-mflat$ flag to gcc.

A=0
BEGIN
A=1
A=2
ABORT

Figure 2: Example Transaction

Example of a simple transaction:

Consider the simple code snippet in Figure 2 where the variable 'A' is initialized to 0 and updated twice inside a transaction. When the control reaches the $begin$, the log-ptr is initialized to the beginning of the log. When 'A' is updated within the transaction, the address of 'A' and the old value at 'A' are written to the log and the log-ptr is updated. Also, the value of 'A' is updated in place in memory. When the control reaches $abort$, the values in the log are replayed and the memory writes are undone. The log-ptr is also updated accordingly to point to the beginning of the log. On the contrary, if a $commit$ is seen the log-ptr is re-initialized to point to the beginning of the log and the memory writes are retained.

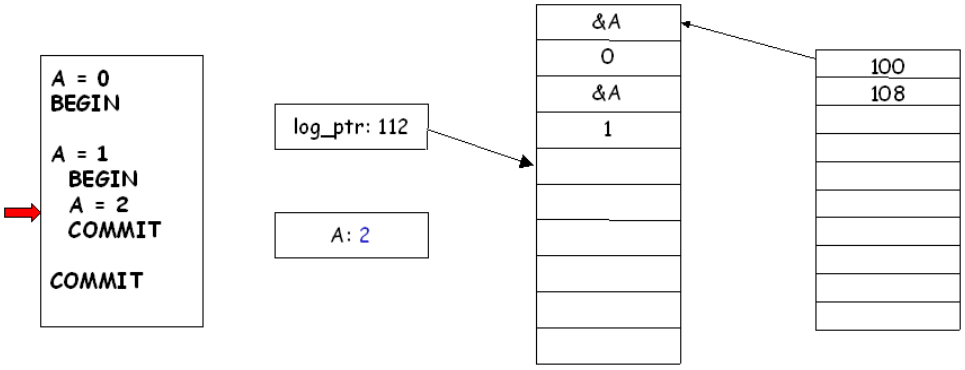


Figure 3: **Example of a nested transaction.** *The state of the system after the statement A = 2.*

3.1 Nested Transactions

Nesting of transactions is necessary for several reasons.

- In the presence of libraries that use transactions, nesting may be required to call procedures in these libraries from another transaction.
- Nested transactions are helpful in minimizing the amount of work undone during a large transaction. Each new portion of the transaction can be written as a nested transaction. This nesting allows us to attempt each portion of the task and undo it when necessary, without having to abort the entire transaction.
- Nested transactions also help in isolating errors in certain operations.

The begin-pointer stack is used to store pointers to the beginnning of each transaction. During each $begin$, the log-ptr is pushed into the begin-pointer stack. During a $commit$ or $abort$, the log-ptr is popped from the

stack. Table 1 discusses the exact semantics of these operations.

BEGIN	1) Push the current value at the log-pointer into the stack 2) Checkpoint the register file
STORE	1) Compute EA 2) write EA and old value at EA to the log 3) increment the log-pointer and store
COMMIT	Pop an address from the stack
ABORT	1) Pop an address from the stack 2) undo writes until the popped address 3) reset the log-pointer to the popped address 4) restore the register file to the last checkpoint

Table 1: Transaction operations

4 Implementation

4.1 Code Snippets

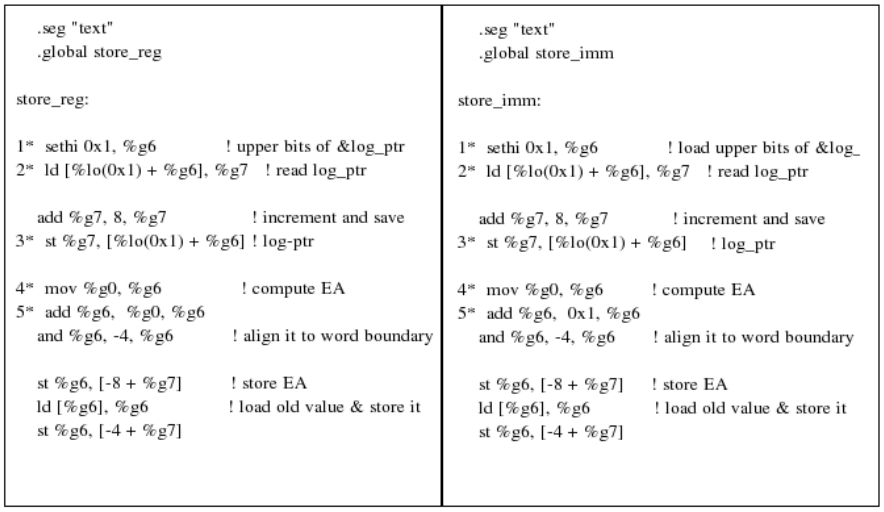
We have written 6 assembly language snippets that will be inserted in the executable. They can be classified into three categories.

1. Initialization Snippet: This snippet initializes the values of the tops for the Begin-pointer Stack (BS) and Register Checkpoint Stack (RCS). It is implemented as a routine and is called once at the beginning of the program.

2. Store snippets: There are 2 formats of store instructions in SPARC: $st\ rd,\ [rs_1 + rs_2]$ and $st\ rd,\ [rs_1 + imm]$. They differ in the way in which the effective address is calculated. There are 2 snippets, one for each format of store instruction.

3. Transaction primitives: There are 3 snippets, one each for *begin, commit and abort*. They perform the operations enumerated in Table 1.

In EEL, it is possible to wrap a snippet as a routine and insert calls to this routine at required program points. We use this approach only for the initialization snippet. The advantage of this approach is that it a full window of 32 registers is available for the snippet. However, it also involves the overhead of a function call every time the snippet is executed. Furthermore, if the snippet uses local information (like rs_1, rs_2, rd, imm etc in case of store snippets), they must be passed to the routine as parameters. This leads to unnecessary



```
   .seg "text"
   .global store_reg

store_reg:

1*  sethi 0x1, %g6            ! upper bits of &log_ptr
2*  ld [%lo(0x1) + %g6], %g7   ! read log_ptr

   add %g7, 8, %g7             ! increment and save
3*  st %g7, [%lo(0x1) + %g6] ! log-ptr

4*  mov %g0, %g6              ! compute EA
5*  add %g6,  %g0, %g6
   and %g6, -4, %g6           ! align it to word boundary

   st %g6, [-8 + %g7]         ! store EA
   ld [%g6], %g6              ! load old value & store it
   st %g6, [-4 + %g7]
```

a. reg-reg addressing

```
   .seg "text"
   .global store_imm

store_imm:

1*  sethi 0x1, %g6            ! load upper bits of &log_
2*  ld [%lo(0x1) + %g6], %g7   ! read log_ptr

   add %g7, 8, %g7             ! increment and save
3*  st %g7, [%lo(0x1) + %g6]    ! log_ptr

4*  mov %g0, %g6          ! compute EA
5*  add %g6,  0x1, %g6
   and %g6, -4, %g6          ! align it to word boundary

   st %g6, [-8 + %g7]        ! store EA
   ld [%g6], %g6             ! load old value & store it
   st %g6, [-4 + %g7]
```

b. reg-imm addressing

Figure 4: **Store snippets**

saving and restoring of the out registers if they are live at that point. The flipside of not wrapping snippets into routines is that only two registers $\%g6, \%g7$ are available without any save-restore overhead.

The two Store snippets are given in Figure 4. Other snippets are given in the Appendix A. EEL provides tagged snippets, which extend ordinary snippets with names to reference individual instructions in a snippet. Both the snippets have 5 tagged instructions. While creating a snippet for the instructions $st\ rd, [rs_1 + rs_2]$ and $st\ rd, [rs_1 + imm]$,

1. 0x1 in instruction 1 is patched with %hi(log_ptr)
2. 0x1 in instructions 2, 3 is patched with %lo(log_ptr)
3. %g0 in instruction 4 is patched with rs_1
4. %g0 in instruction 5 is patched with rs_2 for reg-reg addressing
5. 0x1 in instruction 5 is patched with imm for reg-imm addressing

The size of the old value written to the log is a word even for instructions (stb and sth) that store bytes and half-words. So, we align the effective address to word boundary[1].

[1] instruction $and\ \%g6,\ -4,\ \%g6$ after tagged instruction number 5

procedure dfa()

1: $worklist \leftarrow \Phi$
2: **for all** basic blocks b **do**
3: $b.level_{in} = b.level_{out} = 0$
4: $b.level_{body} \leftarrow$ no. of $begin$'s in b - no. of $commit/abort$'s in b
5: $worklist$.insert(b)
6: **end for**
7: **while** worklist is not empty **do**
8: pop an element b from $worklist$
9: $b.level_{in} \leftarrow \max_{p \in pred(b)} p.level_{out}$
10: $b.level_{out} \leftarrow b.level_{out} + b.level_{body}$
11: **if** $b.level_{out}$ has changed **then**
12: insert b's successors into worklist
13: **end if**
14: **end while**

Figure 5: Dataflow analysis

4.2 What stores to log?

A Store snippet is inserted before a store instruction either

1. if it appears in a routine that is called directly or indirectly from inside a transaction or
2. if there is some path from the routine entry block to this instruction such that the nesting level of the block is greater than zero.

Stores that satisfy condition 1 are identified using call-graph analysis and is discussed in section 4.3. Identifying the nesting level of stores in a basic block is simple once we compute the nesting level of the first instruction of the basic block. We use dataflow analysis [aho, ullman, sethi] to determine the nesting level of a basic block. A worklist algorithm is used to perform this analysis. In this algorithm, we make an assumption that a transaction started within a loop iteration will commit or abort before the end of the iteration. i.e, it is not possible to create a transaction in one iteration of the loop and end it in another iteration of the loop or outside the loop. Without this assumption, the algorithm will not terminate. The algorithm dfa to perform dataflow analysis is given in Figure 5.

4.3 Function calls within Transactions

If procedures can be called from a transaction, store instructions within the called procedure must be logged. For e.g., consider the call graph of a program, given in Figure 6. Since functions $f1$ and $f2$ can be called from a transaction, stores in $f1$ and $f2$ must be logged. Function $f3$ is never called from inside a transaction.

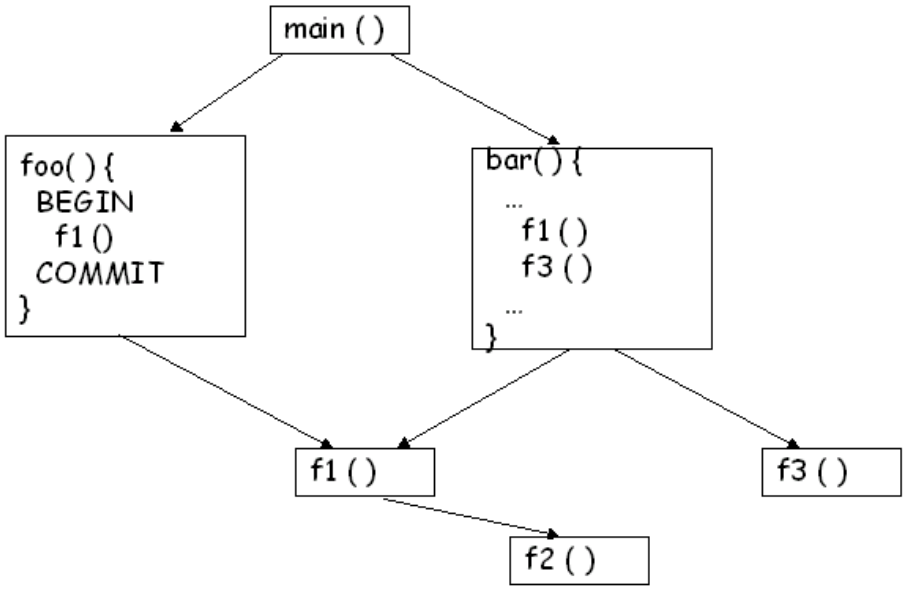


Figure 6: **A simple call-graph** *Stores in functions f1 and f2 get logged and stores in f3 are not logged*

So, stores in $f3$ need not be logged. The algorithm to find whether a function can be called from inside a transaction is similar to dataflow analysis.

5 Evaluation

This section discusses the evaluation of our system. Section 5.1 discusses how we verified the correct implementation of the semantics of $abort$ and $commit$. Section 5.2 discusses performance overhead as a result of logging.

5.1 Correctness

We choose two encryption algorithms, Message Digest (MD5) and Secure Hashing Algorithm (SHA) to verify the correctness of our implementation. These algorithms do bitwise operations (like xor, and etc) on their input. Thus, they are very sensitive to changes even in a single bit of their inputs. Consequently, even a minor error in their input will produce a completely different output.

- **Verifying Abort:** The simplest way to verify $abort$ is to call the encryption function from inside a transaction and aborting the transaction. In this case, no encryption happens and the program state is

restored to the initial checkpoint. We also verify this behavior by aborting at different locations within the encryption procedure as well as after the completion of encryption.

- **Verifying Commit:** The simplest way to verify $commit$ is to call the encryption function from inside a transaction and committing the transaction. In this case, the output is the same as a call to the encryption function without a transaction. Also, we abort the transaction randomly at different locations and restart the transaction. In this scenario too, we get the correct output of encryption.

- **Verifying Nested Transactions:** Nested transactions are verified using a matrix multiplication program. The multiplication of two 10x10 matrices is performed within a transaction. In addition, the multiplication of a row with a column is done inside an inner transaction. The inner transactions are aborted randomly and the outer transaction is committed. In the output matrix, elements corresponding to the inner transactions that aborted had a zero (which was the value before the transaction) and elements corresponding to committed transactions had the correct output.

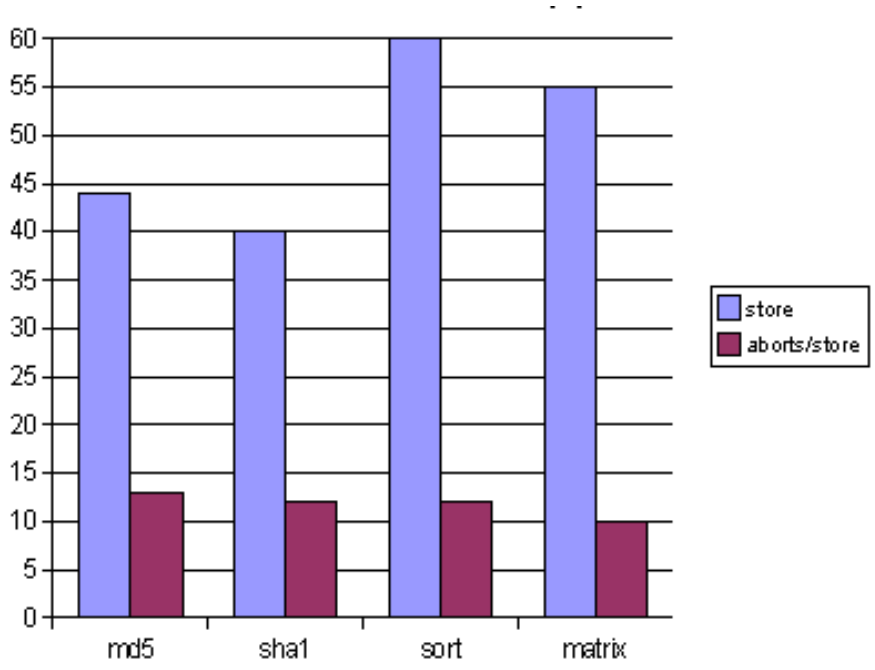


Figure 7: **Overhead for different applications**

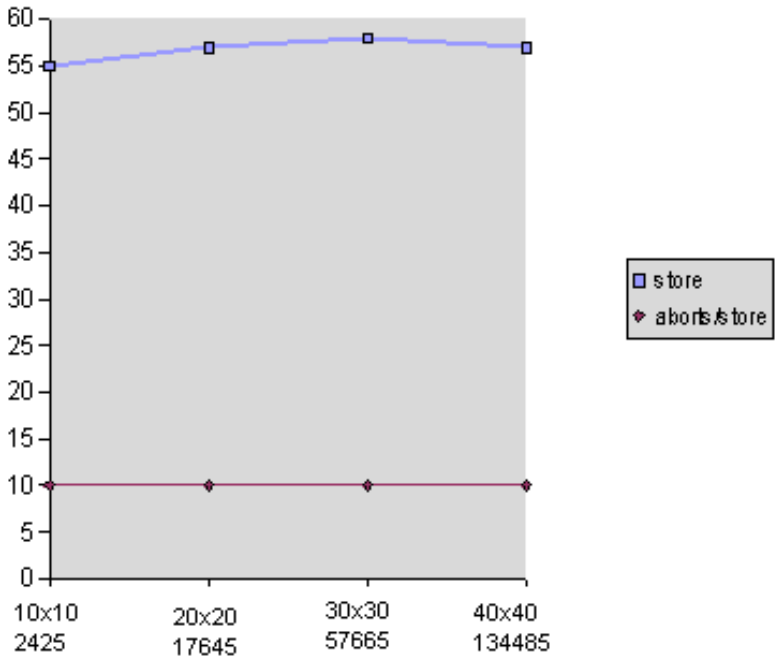


Figure 8: **Overhead for different number of stores** *The second number along the x-axis shows the number of store entries created during the matrix multiplication.*

5.2 Performance

Transactions are aimed at simplifying error recovery (and parallel programming). However, it also incurs an overhead because of logging. The use of transactions involves a trade-off between performance and ease of programming. It is very difficult to quantify the gain obtained by simplifying program development. So, we measure the overhead incurred and justify that it is reasonable. We measure the performance overhead on four programs: MD5, SHA, quicksort and matrix-multiply.

Figure 7 shows the number of cycles taken to create and abort a log entry. On an average, creating a log entry takes 50 cycles and undoing an entry takes 11 cycles. The reason behind the larger number of cycles to create a log could be that the log-ptr is read, incremented and stored during each creation. On the other hand, access to the log-ptr takes place only once during a transaction abort and is amortized over a large number of undo operations. Figure 8 shows the variation of store and abort overhead for various sizes of the logs. The size of the logs is varied by varying the dimensions of the matrices multiplied. The per-log creation and abort overheads stay constant for increasing log sizes. This shows that our implementation scales well with large log sizes. Figure 9 shows the overhead due to transactions in terms of the overall execution time. The

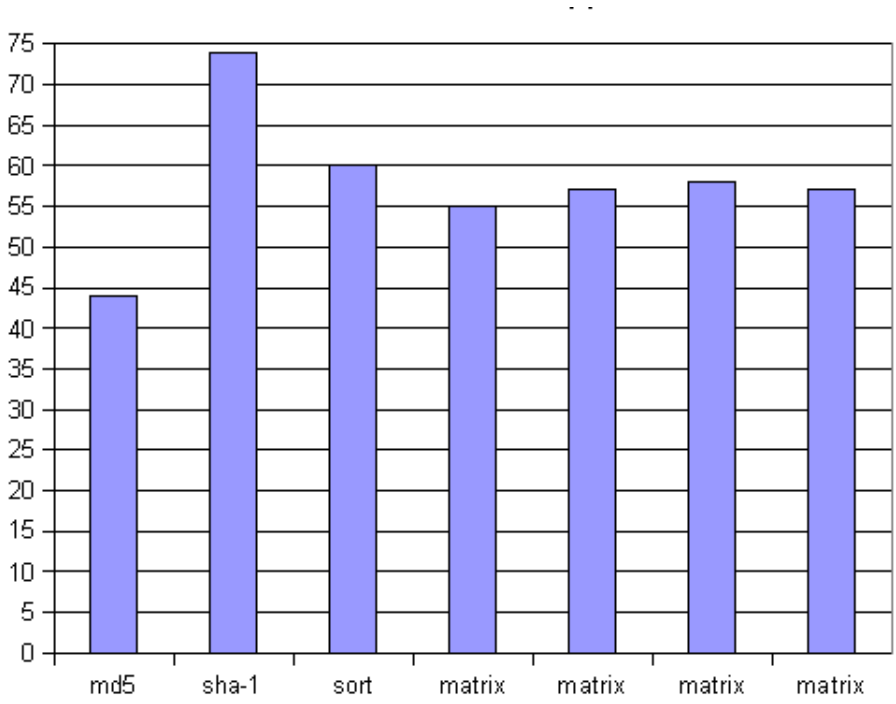


Figure 9: **Slowdown for different applications***The y-axis in the graph is the ratio of overhead of logging to the original execution time.*

y-axis in the graph is the ratio of overhead of logging to the original execution time. For e.g., a value of 1 in the y-axis means that overhead of logging is the same as the original exection time. i.e. the new program executes at half the original speed. MD5 and SHA incur considerably higher overhead because they perform very few computations and spend most of their execution time in store operations. On the other hand, matrix multiply and sort perform other computations and hence spend a smaller portion of their execution time in store operations.

The overhead incurred is reasonable and can be tolerated for two reasons. Firstly, the entire program will not be inside a transaction and hence the overhead will not be severe. Secondly, the 90-10 rule in software engineering says that programs spend 90% of their execution time in only 10% of the instructions. So, it is reasonable to sacrifice performance in the remaining 90% code for ease of programming and correctness.

6 Future Work

6.1 Minimizing Store Overhead

- **Minimizing the number of stores:** In the current implementation of the system, a function is assumed to be inside the transaction if it is called from within a transaction in atleast one of its callers. In the call graph given in Figure 6 the stores in function $f1$ and $f2$ will be logged irrespective of whether it is called from function foo or bar. An alternative would be to use two versions of a function: one that is implemented as a transaction and another version that is not. This eliminates logging stores unnecessarily when it is not called from inside a transaction. Such duplication can also be done at the level of a basic block.

- **Minimizing the overhead per store:** The overhead per store can be minimized by decreasing the number of times the log-ptr is updated. Our implementation updates the log-ptr after every store. This means the log-ptr is loaded from memory, incremented and stored back to memory after every store. Instead, we can group the stores within a basic block and update the log-ptr once for all the successive stores. This would minimize the overhead incurred per store.

```
mov 0, %r1
begin
...
ld [a],%r1
abort
div %r2,%r1,%r3
```

Figure 10: **Example of a transaction in the presence of compiler optimization** *The use of transactions in the presence of compiler optimizations might lead to incorrect execution.*

6.2 Implementing log-based transactions in Compilers

Implementing log-based transactions in compilers helps improve performance. Our implementation has some issues working with optimized programs. Consider the simple example given in Figure 10 where the live range of register %r1 crosses a transaction boundary. The presence of transactions in such scenarios might lead to incorrect execution. Implementing the semantics of log-based transactions in compilers provide the following advantages

- The code snippets for store, commit, begin and abort can take advantage of compiler optimizations like register allocation. This would decrease the per store overhead.
- Correct execution can be ensured in scenarios like the one shown in Figure 10.

Acknowledgements

We thank Prof. Mark Hill for giving us a wonderful opportunity to work on an interesting research project. We thank Matthew Allen for his valuable guidance on issues related to EEL. We also thank Prof. Mark Hill and students of the Multifacet group for helping us during the various phases of the project.

7 Conclusion

Concurrent programming is witnessing a paradigm shift due to the evolution of Transactional memories. In this project, we develop a log-based transaction system for single threaded applications that is based on LogTM. We measure the performance overhead incurred in our system. We conclude that the ease of programming achieved overshadows the overhead incurred.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture*, pages 316–327. Feb 2005.

[2] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, Jun 2004.

[3] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimising memory transactions. *to appear in the proceedings of PLDI '06*, 2006.

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

[5] J. R. Larus. Eel Guts: Using the eel executable editing library. 1996.

[6] J. R. Larus and E. Schnarr. Eel: machine-independent executable editing. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 291–300, New York, NY, USA, 1995. ACM Press.

[7] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. Logtm: Log-based transactional memory. *International Symposium on High Performance Computer Architecture (HPCA)*, 2006.

[8] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM Press.

[9] A. Shinnar, D. Tarditi, M. Plesko, and B. Steensgaard. Integrating support for undo with exception handling. Technical report, December 2004.

A Appendix

A.1 Store Snippets

```
 !!
 !! store_reg create an entry in the log for a store register instruction
 !!

.seg "text"
.global store_reg

store_reg:

1*  sethi 0x1, %g6                  ! upper bits of &counter
2*  ld [%lo(0x1) + %g6], %g7         ! load counter

add %g7, 8, %g7
3* st %g7, [%lo(0x1) + %g6]

4* mov %g0, %g6
5* add %g6,  %g0, %g6
and %g6, -4, %g6

st %g6, [-8 + %g7]
ld [%g6], %g6
st %g6, [-4 + %g7]

 !!
 !! store_imm create an entry in the log for a store for a store immediate instruction
 !!

.seg "text"
.global store_imm

store_imm:

1*  sethi 0x1, %g6                  ! upper bits of &counter
2*  ld [%lo(0x1) + %g6], %g7         ! load counter

add %g7, 8, %g7
3* st %g7, [%lo(0x1) + %g6]

4* mov %g0, %g6
5* add %g6,  0x1, %g6
and %g6, -4, %g6

st %g6, [-8 + %g7]
ld [%g6], %g6
st %g6, [-4 + %g7]
```

A.2 Initialization Snippet

```
  !! Modifies: (no visible changes)

.seg "text"
.global init


init:

save %sp, -96, %sp
mov %g1, %l3 ! No visible changes

mov 0x0, %g7
1* sethi 0x1, %g7
2* or %g7, 0x1, %g7
st %g7, [%g7 - 4]

    mov 0x0, %g7
3* sethi 0x1, %g7
4* or %g7, 0x1, %g7
st %g7, [%g7 - 4]

    mov 0x0, %g7
5* sethi 0x1, %g7
6* or %g7, 0x1, %g7
st %g7, [%g7 - 4]

mov %l3, %g1 ! No visible changes
ret
restore
```

A.3 Commit Snippet

```
 !! Modifies: (no visible changes)

.seg "text"
.global commit


commit:

1*  sethi 0x1, %g6                  ! upper bits of &stack_top
2*  ld [%lo(0x1) + %g6], %g7        ! load stack_top

add %g7, -4, %g7
3* st %g7, [%lo(0x1) + %g6]

4* add %g6, 0x1, %g6
```

```
add %g7, -4, %g7

cmp %g6, %g7
bne L15
nop

mov 0x0, %g7
5* sethi 0x1, %g7
6* or %g7, 0x1, %g7
st %g7, [%g7 - 4]

L15:
7*  sethi 0x1, %g6                  ! upper bits of &checkpt_top
8*  ld [%lo(0x1) + %g6], %g7        ! load checkpt_top

9* sub %g7, 0x0, %g7 ! subtract checkpt_size
10* st %g7, [%lo(0x1) + %g6]
```

A.4 Begin Snippet

```
 !! Modifies: (no visible changes)

.seg "text"
.global begin


begin:

1*  sethi 0x1, %g6                  ! upper bits of &counter
2*  ld [%lo(0x1) + %g6], %g7        ! load counter

3*  sethi 0x1, %g6                  ! upper bits of &stack_top
4*  ld [%lo(0x1) + %g6], %g6        ! load stack_top

st %g7, [%g6]
add %g6, 4, %g6

5* sethi 0x1, %g7
6* st %g6, [%lo(0x1) + %g7]

7*  sethi 0x1, %g6                  ! upper bits of &checkpt_top
8*  ld [%lo(0x1) + %g6], %g7        ! load checkpt_top

st %i0, [0 + %g7]
st %i1, [4 + %g7]
st %i2, [8 + %g7]
st %i3, [12 + %g7]
st %i4, [16 + %g7]
st %i5, [20 + %g7]
```

```
st %i6, [24 + %g7]
st %i7, [28 + %g7]

st %o0, [32 + %g7]
st %o1, [36 + %g7]
st %o2, [40 + %g7]
st %o3, [44 + %g7]
st %o4, [48 + %g7]
st %o5, [52 + %g7]
st %o6, [56 + %g7]
st %o7, [60 + %g7]

st %g0, [64 + %g7]
st %g1, [68 + %g7]
st %g2, [72 + %g7]
st %g3, [76 + %g7]
st %g4, [80 + %g7]
st %g5, [84 + %g7]
st %g6, [88 + %g7]
st %g7, [92 + %g7]

st %l0, [96 + %g7]
st %l1, [100 + %g7]
st %l2, [104 + %g7]
st %l3, [108 + %g7]
st %l4, [112 + %g7]
st %l5, [116 + %g7]
st %l6, [120 + %g7]
st %l7, [124 + %g7]

9* add %g7, 0x0, %g7 ! add checkpt_size
10* st %g7, [%lo(0x1) + %g6]
```

A.5 Abort Snippet

```
 !! Modifies: (no visible changes)

.seg "text"
.global abort

abort:

mov 0x0, %g7
1* sethi 0x1, %g7
2* or %g7, 0x1, %g7

ld [%g7], %g2

mov 0x0, %g6
```

```
3*  sethi 0x1, %g6              ! upper bits of &stack_top
4*  or %g6, 0x1, %g6          ! load stack_top

ld [%g6], %g1
add %g1, -4, %g1
st %g1, [%g6]

ld [%g1], %g5
add %g1, -4, %g1


L0:
cmp %g2, %g5
be L1
nop

ld [%g2 - 4], %g3
ld [%g2 - 8], %g4
st %g3, [%g4]

    sub %g2, 8, %g2
cmp %g0, %g0
be L0
    nop


L1:
st %g5, [%g7]
cmp %g1, %g6
bne L2

mov 0x0, %g7
5* sethi 0x1, %g7
6* or %g7, 0x1, %g7
st %g7, [%g7 - 4]

L2:

7*  sethi 0x1, %g6                  ! upper bits of &checkpt_top
8*  ld [%lo(0x1) + %g6], %g7        ! load checkpt_top
9* sub %g7, 0x0, %g7          ! subtract checkpt_size

ld [0 + %g7], %i0
ld [4 + %g7], %i1
ld [8 + %g7], %i2
ld [12 + %g7], %i3
ld [16 + %g7], %i4
ld [20 + %g7], %i5
ld [24 + %g7], %i6
ld [28 + %g7], %i7
```

```
ld [32 + %g7], %o0
ld [36 + %g7], %o1
ld [40 + %g7], %o2
ld [44 + %g7], %o3
ld [48 + %g7], %o4
ld [52 + %g7], %o5
ld [56 + %g7], %o6
ld [60 + %g7], %o7

ld [64 + %g7], %g0
ld [68 + %g7], %g1
ld [72 + %g7], %g2
ld [76 + %g7], %g3
ld [80 + %g7], %g4
ld [84 + %g7], %g5
ld [88 + %g7], %g6
ld [92 + %g7], %g7

ld [96 + %g7], %l0
ld [100 + %g7], %l1
ld [104 + %g7], %l2
ld [108 + %g7], %l3
ld [112 + %g7], %l4
ld [116 + %g7], %l5
ld [120 + %g7], %l6
ld [124 + %g7], %l7


10* st %g7, [%lo(0x1) + %g6]
```