Extending Open64 with Transactional Memory Features

Jiaqi Zhang Wenguang Chen Weimin Zheng
Tsinghua University
zhang-jq06@mails.tsinghua.edu.cn, {cwg, zwm-dcs}@tsinghua.edu.cn

Abstract

The fast development of parallel platforms is demanding more parallelism in modern applications. However, the manipulation of mutual-excluded memory accesses is obstructing the way towards high productivity in parallel software development for shared memory system. Transactional Memory (TM) is a promising paradigm that helps abstract the complexity of concurrency while keeping the scalability. And the compilers for TM are needed in order to facilitate both parallel programmers and TM researchers. This paper describes the design, detailed implementation, and optimization of our extension of TM features in Open64. The preliminary experimental results show that our optimized implementation of the compiler, together with a high performance TM runtime, have the potential to be competitive for parallel programs with fine grained locks.

1. Introduction

With the dominance of multi-core computers, parallel programming models have been a hot topic. In order to better utilize the ever increasing cores, efforts are made to explore concurrency in real world applications. Message passing interface emphasizes the isolation between the data in each process. However, it needs the programmers to organize all the communications including timing, content, targets etc. and is therefore difficult to program. The shared memory interface is much more intuitive as all the working threads work on the same contents. Nevertheless, the use of locks to insure exclusive access tends to be problematic. Coarse-grain locks may prohibit possible concurrency and produces serialized code; on the other hand, fine-grain locks introduce complexity in their management, and thus may incur software engineering problems such as deadlock. In addition, lock based applications are difficult to exhibit composability, which is the major way to perform code reuse in order to increase productivity.

Transactional Memory (TM) is a promising paradigm that helps produce a composable[11] parallel program. It simplifies the manipulation of critical regions by abstracting away the complicated lock acquirement and release, and providing a clear interface that marks the atomic blocks. The isolation property of transaction permits the same memory location to be appeared in different transactions without the programmers' annotation. To bring the parallelism, TM deploys an optimistic strategy that allows concurrent accesses to the shared memory, and aborts the transactions on conflicts. Thus there are opportunities that all the transactions are successfully executed even when they are updating the same chunk of memory, only if different parts are accessed.

There are several Software Transactional Memory implementations available, for example, DracoSTM[1], TinySTM[2], TL2[3], and LibLTX, etc.. However, they are all provided as libraries, and it is obviously difficult to manually instrument all the accesses to shared locations, especially when the transactions are large. Thus automatic transformation tools or compilers are needed to facilitate the use of STM.

Some experimental STM compilers are also

available, such as the Mercurium OpenMP compiler[4], the Intel C++ STM compiler[5], OpenTM compiler[6], and TANGER[7]. The Intel C++ STM compiler is the first production level STM compiler. It is based on the high performance Intel Compiler, and it implements the transactional memory algorithms and translation techniques described in [8] and [9]. It supports atomic blocks in both Pthreads and OpenMP by the generic directive **#pragma tm_atomic**, and is heavily optimized to reduce the transactional memory overhead. The most recent version of the compiler also supports user specified transactional memory library implementation in binary form. However, the source code of the compiler is inaccessible as it is based on a commercial product. Therefore, while it is very useful for the parallel programmers to apply TM to their applications, the TM runtime developers are unable to understand the corresponding compiler infrastructure. Since it is impossible to modify the code of the compiler, it also limits the runtime developers to the existing interface, which might obstruct the features of the innovative underlying runtime. The OpenTM compiler focuses on extending OpenMP with Transactional Memory semantics. It provides extensions to the directives so that the OpenMP programs could utilize TM seamlessly. For example, they proposed the basic transaction boundary **#pragma omp transaction**, and also **#pragma omp tranfor** to indicate that the parallelized iteration bodies to be executed as transactions. The Mercurium OpenMP compiler performs similar work, with several further extensions to the attributes of the atomic block. TANGER supports TM instrumentation based on the LLVM compiler. It targets on the LLVM intermediate representation language, and provides TM semantics in both the word-based and object-based manner[12]. During the preparation of this paper, we have found a similar work just published in GCC[10]. As we have not accessed their code, and the paper is mainly about the research highlights instead of implementation and design details, we have insufficient information to perform a comparison here.

Our aim is to provide an open-sourced research infrastructure for transactional memory on the base of Open64, which could facilitate both the TM runtime developers by providing opportunities to integrate their implementations to the compiler seamlessly, and the parallel programmers by presenting a clear transactional memory interface. This aim leads us to implement a uniform interface for both Pthreads and OpenMP, and to perform automatic optimizations as much as possible.

The paper is organized as follows: Section2 describes our design of the compiler. Section3 introduces our implementation and optimization, together with the open issues in the corresponding parts. Section4 presents the results of our preliminary experiments. Section5 concludes the paper and highlights possible future works.

2. Design

2.1. Programming Interface

The basic components of transactional memory semantics are the atomic regions. The atomic regions ensure that all the inner statements are executed atomically, that is, executed all together or not executed at all. Our implementation supports the atomic region by means of **Implicit Transaction**s. The users only specify the boundaries of the transactions instead of annotating any memory access, and all the statements within the boundaries are implicitly executed as a transaction, with the atomicity and isolation being guaranteed automatically. This approach could greatly benefit the programming productivity by easing the programmers the burden of utilizing transactional memory semantics, and also make the target programs clear to read. It also enables easy reuse of library

functions, since the programmers do not need to consider inserting barriers to them in the new context.

To achieve this goal, we introduce the atomic construct to mark the boundaries of atomic regions. The syntax is

```
#pragma tm atomic [clause]
 structured block
```

where *clause* is optional, and can be one of the following: **readonly**, **shared (list)**, or **private (list)**. The structured block could include any possible statement, including function calls. The compiler automatically instruments the code by replacing memory access operations with function calls to read/write barriers. The underlying transactional memory library validates the read/written data and rolls back to the beginning of the block on conflicts. The programmers are allowed to optionally mark the variables that need or are free of transactional instrumentations by specifying the *shared* or *private* clauses. The compiler does not perform transactional bookkeeping for *private* variables. And if the *shared* clause is specified, only the listed variables are to be watched. As some STM runtime, for example, TL2 and TinySTM, provide particular optimizations for read only transactions, the compiler also provides an opportunity for the programmers to specify the **readonly** clause to indicate that there is no write to shared memory in the transaction. The intent of these clauses is to help decrease function calls and thus increase performance. But the programmers should make it in their own risk as data races on them could not be detected.

While the conflicts between transactions can be handled by the runtime, our system does not support isolation between transactional accesses and non-transactional accesses. Thus it is a **weak isolation** system, which means the memory accesses out of transactions are not logged and thus its conflicts, even with transactional accesses, cannot be detected. So the design requires the programmers to make sure that all the accesses to shared memory are included into atomic regions, otherwise the consistency cannot be guaranteed. This approach is chosen because the transactional bookkeeping of non-transactional accesses is too expensive for STM implementations. All single accesses outside the transactions should be marked as a micro transaction if strong isolation should be guaranteed. However, the overhead is unacceptable. And the optimizations discussed in Section3.3 are not always feasible given the complex nature of unmanaged code. Many unsolved problems, alias analysis for example, have obstructed the success of automatic detections of shared data.

During the execution of a transaction, the user could abort the transaction at any time by the abort construct:

```
#pragma tm abort
```

It rolls back the current thread to the entry of the transaction and re-executes it. This construct is specifically useful when the transaction is waiting for certain statuses such as flags.

Two more constructs are introduced to mark the functions that could be called inside transactions. Their syntaxes are:

```
#pragma tm function
function-declaration
#pragma tm waiver
 function-declaration
```

A function that is marked with *tm function* could be invoked in atomic regions with guaranteed consistency. These functions are referred to as **transactional functions**. The memory accesses within a transactional function are also instrumented with read/write barriers, and the runtime keeps transactional bookkeeping for all the shared variables the function accesses. When the validation fails inside the transactional function, the thread rolls back to the entry of the enclosing transaction and re-executes it.

A function that is marked as *tm waiver* could also be invoked in atomic regions. However, there is no instrumentation for the inner memory

Table1 The Transactional Memory interface provided by TL2

Interface	Description
Thread* TxNewThread()	Allocate a new Thread structure to keep logs
TxStart(Thread* Self, jmp_buf* buf, int flags)	Start a new transaction for current thread
TxCommit(Thread* Self)	Commit the current transaction
TxLoad(Thread* Self, void* addr)	Perform synchronized load from given memory address
TxStore(Thread* Self, void* addr, intptr_t val)	Perform synchronized store to given memory address
TxStoreLocal(Thread* Self, void* addr, intptr_t val)	Perform locally logged store to given memory address
TxAbort(Thread* Self)	Abort the current transaction and re-execute

accesses, even those of shared variables. While the compiler would issue warnings on the memory access to possible shared variables, the programmers should not purely rely on it. We refer to these functions as **transaction-free function**s. It is similar to the *pause* semantic. When the transaction invokes these functions, all the transaction related behaviors such as logging and validations pauses, and they resumes when the function returns. It could be used in order to avoid unnecessary calls to runtime libraries when the programmers are aware of the risk of data races.

Only transactional functions and transaction-free functions are allowed to be invoked in atomic regions. Calls to any other functions, including legacy code (for example, the binary library functions), would issue compilation errors, as they may bring data races without the programmers' awareness. To utilize the legacy code in compiled libraries, binary instrumentations may be performed. [7] and [14] make attempts to achieve the goal by either applying Dynamic Binary Rewriting or disassembling the binary code. An insight discussion is out of the scope of this paper.

2.2. STM runtime library

The implementation targets on word-based STM libraries, and it currently adopts TL2 as the underlying runtime library. TL2 is a fast STM library with concise interfaces, which are shown in Table1 together with their descriptions. It is originally developed by SUN, and is ported to i386 and X86_64 platforms by Stanford. A detailed description of the library could be found in [3].

While most of the functions are typical transactional memory interfaces, *TxStoreLocal* is an assistant function that aims to optimize the programs. It targets on thread local variables that only need to be rolled back on transaction abortion, but not to be synchronized.

Note that while TL2 is used in the current implementation, the runtime library could be replaced with slight or even no modifications to the compiler if the interfaces are similar. The compiler is designed to be loosely coupled with the runtime library, and a set of wrapper functions are provided to adapt to different interfaces. Therefore, it is much easier for STM library developers to integrate their innovative implementations to the compiler. For example, TinySTM library could be integrated with only slight modifications to the wrapper functions instead of the inside of compiler. And it is also easy to further extend the compiler to support extra features of TinySTM such as registration of event handlers, given the existing infrastructure.

2.3. Optimization Opportunities

Unlike exclusive regions protected by locks which need to be as small as possible, transactions tend to be large and composable in order to better facilitate the programmers. Thus while it is much slower than hardware implementation of TM, STM is a hot topic given its much lower cost in development and unbounded nature. Since it is very possible that there are numerous statements in a single

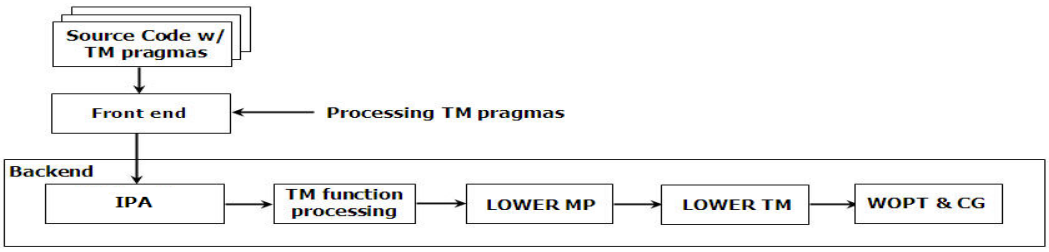

Fig.1. The phases for lowering TM in the backend of Open64

transaction, one of the main overhead for STM programs is the calls to synchronized read and write. It is especially true for automatically generated code, where there are lots of unnecessary calls to the STM runtime library. So the main purpose of our optimization is to eliminate them as much as possible. However, when dealing with indirect accesses via pointers, a conservative design is adopted as it is difficult to determine whether it is accessing shared data due to alias problems.

3. Implementation

3.1 General Transformation

The transformation of the original program to TM code is implemented as a standalone lowering phase in the backend of Open64 as illustrated in Figure 1, which presents the related or remarkable phases in Open64, and marks the relative position of the TM lowering. The main transformation takes part in the LOWER TM phase, and the TM function processing clones the transactional functions described in Section3.2.

In order to perform rollback, the runtime requires a jump_buf being set on the entry of the transaction. At the beginning of the atomic region, the compiler generates the call to set the jump_buf and passes it to TxStart(). After that, all the statements in the block are checked one by one, including their kids, and are replaced by STM runtime library functions if necessary. Note that WHIRL does not permit nested statements, and thus simple replacement of leaf node such as LDID with library functions would not produce a legal WHIRL tree. In our implementation, the call is performed in a generated statement node, and the returning value is stored via the mapping mechanism, so that the ancestor could get it by generating new LDID to it. The result of an assignment statement is presented by Figure 2, where the right column is the rough WHIRL tree representing the left statement.

The transformation of expressions is similar to the process with the exception that the returning value of TxRead should be saved to another register, as there might be multiple leaves which require the synchronized read.

For conditional jump control flow statements such as IF, their bodies are processed in the same way as above. However, to correctly handle their boolean expressions, the synchronized read should be performed ahead of the IF statement.
For loop structures such as WHILE_DO, besides the above process, the read barrier of loop variant should also be inserted to the end of the loop bodies so that the loop condition could be validated in every iteration. To ensure the correctness of nested control flow statements, a stack is used to save the generated synchronized read of loop variants, and is popped at the end of loop bodies.

While structured statement block is adopted for a transaction in our design in order to keep the atomic block clear, jump statements are allowed. If a statement such as break jumps out of the transaction, it incurs a transaction commit. And the transaction is re-executed instead of escaped if it fails to commit.

The result of a general transformation of original TM marked code is shown in Figure 3. The left column is a code segment in a parallel

a = b;	LDA <b> PARM VCALL <TxRead> #synchronized read of b LDID <return_offset> PARM LDA <a> PARM VCALL <TxWrite> #synchronized write to a

Fig.2. The transformed WHIRL tree for a single assignment statement

task, where result is a global shared variable that is expected to be accumulated in each thread. In order to be concise and clear, a psudo-code is listed instead of the actual WHIRL tree. The Self variable in the right column is used to manipulate the execution information of the transaction, and is generated by library function TxNewThread at the entry of the thread. For OpenMP, the function is automatically invoked at the stage of forking threads. For Pthreads, the programmers currently would have to manually invoke a wrapping function tm_thread_start at the start of the parallel task, which needs no argument and handles all the initializations of the thread local logging.

3.2 Functions

To support function calls in transactions, two directives are introduced as described in Section2.1. The front end marks the functions with corresponding attributes when the directives are seen. And as illustrated in Figure 1, the transactional functions are processed at the early stage of the backend processing. For the transactional ones, the compiler clones the whole function, gives the cloned function a mangled name which is the original name preceded with fixed prefix that would not incur confusions, and records it in the transactional function table. The cloned function is lowered right after the original one, and is instrumented with read/write barriers in the same way as the atomic regions, with the exception that it doesn't have to start the transaction or set up the jump buffer itself. For the transaction free functions, except for setting up the corresponding attributes in its ST, no special treatment is performed.

When a call to function is seen in an atomic block, the compiler first checks its attributes. If it is a transaction free function, the lowering phase continues. If it is a transactional function, the compiler checks the transactional function table with the mangled name. If it fails to find the function, an error is issued. Otherwise, the function call is replaced with the call to the cloned function. The original function is only used when called outside of the atomic blocks.

The transactional and transaction free functions could call each others. However, any unmarked function, including legacy function, is not allowed to be invoked inside them. If a transaction free function is called inside a transactional function, it behaves exactly like when it is called inside an atomic block. However, on the other hand, when a transactional function is called inside a transaction free one, instead of the cloned function, the original one without instrumentation is invoked, and thus the transaction semantic is not guaranteed.

Note that currently we are cloning and instrumenting all the transactional functions, despite whether they are actually called inside any transaction or not. While it does not hurt the correctness or performance of the program, it increases the size of the executable. To achieve the goal of only cloning necessary transactional functions, an inter-procedural analysis is needed, and this work is left for our future work. Another issue is the handling of function calls in indirect manners such as via function pointers. It involves alias problems and is not handled in the current implementation. [8] has solved this problem by marking transactional functions and recording the cloned functions' address at a fixed offset from the target address of the function pointer. We are planning to implement it in the future versions.

```
1.1  int i = 0;
1.2  #pragma tm atomic
     {
1.3      int j = 0;
1.4      for(i=0;i<20;i++)
         {
1.5          for(j=0;j<10;j++)
             {
1.6              result++;
             }
         }
     }
```

```
2.1  int i = 0;
2.2   jmpbuf   jbuf;
2.3   _setjmp(jbuf);
2.4   TxStart(Self, jbuf);
2.5   TxStore(Self, &j, 0);
2.6   for (; TxLoad(Self, &i)<20;
          TxStore(Self, &i, TxLoad(Self, &i)+1)){
2.7     for(TxStore(Self, &j, 0); TxLoad(Self, &j)<10;
            TxStore(Self, &j, TxLoad(Self, &j)+1)){
2.8           TxStore(Self, &result, TxLoad(Self, &result)+1);
        }}
2.9   TxCommit(Self);
```

Fig.3. The transformation result of a code segment

3.3 Optimizations

As described in Section2.3, the major goal of our optimization work is to eliminate unnecessary calls to runtime library calls. By observing the programs, there are two kinds of memory accesses that could be optimized with the efforts of both the compiler and the runtime library. Note that except for the following discussed automatic approach to unnecessary function call elimination, programmers could also specify the variables that do not need read/write barriers via the shared/private clauses as described in Section2.1.

The first one is the variables that do not need to be recorded at all. These variables are often the ones that are declared and used inside the atomic blocks. For example, the variable *j* in line1.3 of Figure 3 does not need any instrumentation. It is not seen anywhere outside the atomic block, thus the success or failure of the transaction has no effect on them. We refer to these variables as **transaction-local** variables. While they are uncommon in traditional lock-based critical regions, which need to be kept tiny to ensure concurrency, they are very likely to occur in programs with largetransactions. The compiler automatically detects transaction-local variables in the front end, when parsing the atomic block, and marks the corresponding attributes, which could be retrieved from its ST in the LOWER TM phase. And the backend would not instrument the memory accesses to these variables. Taking the instrumented code in Figure3 as an example, the recognition of *j* would help eliminate the *TxStore* in Line2.5, as well as all the four runtime library calls in Line2.7. The optimized code is shown in the left column of Figure 4.

The second one may not be obvious at the first glance. But they are more common than the transaction-local variables. They are the thread local variables that are updated in the transaction, which means they only need to be rolled back when the transaction fails, but don't need to be synchronized among all the threads. We refer to these variables as **barrier-free** variables. In fact, for the parallel model of Pthreads, most of the variables that are in the same PU as the transaction and could be seen outside of the transactions are barrier-free variables. For example, the variable *i* in Line1.1 of Figure3 is a barrier-free variable. The variables that are specified by the clause *private* described in Section2.1 are also treated as barrier-free variables. Given the nature of these variables, they are detected via their storage classes. Instead of instrumenting write barrier to the relevant store operation, the compiler uses *TxStoreLocal*, which only logs necessary information for rollback without recording them to the write-set. In addition, the read accesses to these variables are free of any record. In the program of Figure3, the recognition of *i* as a barrier-free variable helps eliminate the two *TxLoad* in Line2.6, and replaces the *TxStore* with light-weight *TxStoreLocal*. The optimized code is shown in

```
1.1   int i = 0;
1.2   jmpbuf   jbuf;
1.3   _setjmp(jbuf);
1.4   TxStart(Self, jbuf);
1.5   for (; TxLoad(Self, &i)<20;
          TxStore(Self, &i, TxLoad(Self, &i)+1)){
1.6     for(j=0; j<10;j++){
1.7         TxStore(Self, &result,
                    TxLoad(Self, &result)+1);
        }}
1.8   TxCommit(Self);
```

```
2.1   int i = 0;
2.2   jmpbuf   jbuf;
2.3   _setjmp(jbuf);
2.4   TxStart(Self, jbuf);
2.5   for (; i<20;TxStoreLocal(Self, &i, i+1)){
2.6     for(j=0; j<10;j++){
2.7         TxStore(Self, &result,
                    TxLoad(Self, &result)+1);
      }}
2.8   TxCommit(Self);
```

Fig.4. The optimized code of Figure3. The left column is the one with transaction-local variables eliminated. The right column is the one with barrier-free variables recognized.

the right column of Figure 4.

However, while the recognition of barrier-free variables decreased synchronizations, there are still redundancies. For example, as illustrated by the right column of Figure4, *i* is logged each time in the iteration, which is actually not needed as it just needs to be reversed back to the original value when the transaction restarts. Therefore, in the future, we plan to checkpoint these variables at the entry of the transaction instead of logging each of their updates.

The automatic detection strategy of transaction-local and barrier-free variables is varied in different contexts. When the compiler is processing Pthreads parallel tasks, since the threads don't share variables declared in the parallel function, the variables that are "AUTO", which means they are on stack, are identified as barrier-free. And only variables that are declared in the atomic block belong to transaction-local variables. For the cloned transactional functions however, all the variables (excluding PSTATIC ones) declared in the PU, which means having the storage class of AUTO, are identified as transaction-local as they are implicitly declared in transactions. And there is no barrier-free variable for the cloned function since it could not access the caller which might have those variables. For the transactions in OpenMP parallel regions, the optimization relies on its implementation. In Open64, the parallel tasks are created as nested functions that are called **micro tasks**, and thus it is obvious that the variables declared in PU scope of the micro tasks are barrier-free ones. However, the variables declared in the enclosing PU, while are also in stack, are shared by default, and thus should be also instrumented with both read and write barriers. Note that all the variables that are specified to be *private* in the OpenMP clause have local copies generated by the implementation of OpenMP, so they are automatically identified as barrier-free variables.

Besides optimizing the variable accesses, the compiler also helps detect read-only transactions by recursively checking all the statements. And if all the store operations are performed on either transaction-local or barrier-free variables, the whole transaction is identified to be read-only.

Note that although we have put efforts in the automatic optimization, the conservative design has led to false positives when trying to identify variables that need instrumentations. For example, all indirect memory accesses are instrumented with read/write barriers by default even when it points to the memory that is allocated in the enclosing thread. In these situations, the programmers could simply specify them as *private*, and performance increase is expected as they are also treated as barrier-free variables. The same happens to the detection of read-only transactions: none of the transactions that contain indirect stores to memory is identified as read-only transactions. Therefore, while we are exploring more chances for optimization, the programmers' annotations via the directive clauses are still very important to improve the performance as much as possible, especially for the applications dealing a lot with indirect

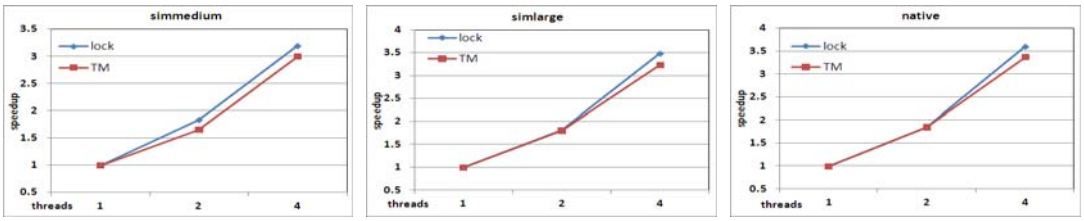

Fig.5. Speedup of *fluidanimate* with both locks and TM using the workload of *simmedium*, *simlarge*, and *native*.

load/stores, as illustrated in section 4.

4. Preliminary Experimental Results

Our preliminary experiment is carried out on an Intel Core2 Quad-Core CPU. The target application are the fluidanimate benchmark in PARSEC1.0[13] and the kmeans benchmark in STAMP[15]. fluidanimate simulates an incompressible fluid for interactive animation purposes with an extension of the Smoothed Particle Hydrodynamics method. It is a fine-grained parallel application with a large working set, and is constituted by five individual kernels. The K-means is a well known data mining algorithm that groups objects in an N-dimensional space into K clusters. We use *fluidanimate* to compare our implementation with fine-grain lock program, and *kmeans* to compare our compiler generated code with the manually instrumented TM program.

Figure 5 shows the comparison between the speedup of the original lock based program and the transactional memory one. Both of them are compiled by Open64. The experiments are done with 3 different input sets provided by PARSEC: the *simmedium*, *simlarge*, and *native*, which contain 100,000 particles, 300,000 particles, and 500,000 particles separately. The speedup is computed over the running time of the serial version of the application.

From the figures, we can conclude that the TM version is not incurring unbearable overhead as it is not slowed down much when running with single thread. At the same time, it resembles the speedup of the original parallel program especially when the input set is larger. More importantly, it helps to largely simplify the concurrency management, especially when it is parallelized in a fine grained manner. For example, when running 4 threads with the input set of simlarge, the original program creates 308224 locks in a two-dimension lock array.

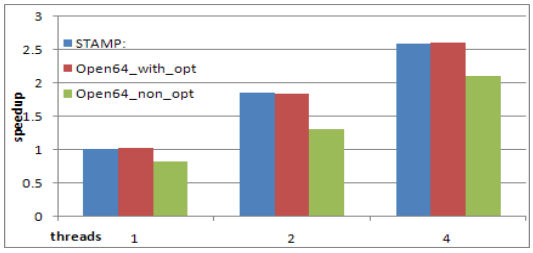

Fig.6. speedup of *kmeans*

Figure 6 shows the speedup of kmeans in three situations: the original manually instrumented version in STAMP, and the Open64 compiled version with/without manual optimizations. Here the manual optimization refers to the user specified clauses discussed in Section 2. The speedup is calculated based on the execution time of the STAMP version with single thread. It shows that without the user specified clause, the performance could be significantly decreased. The following figure illustrates the related code segment.

```
#pragma tm atomic private(feature)
{   int j;
    *new_centers_len[index] ++;
    for(j=0;j<nfeatures;j++){
        new_centers[index][j]+=feature[i][j];
}}
```

Fig.7. Code segment of kmeans

Figure 7 is a core transaction that costs a lot of time during the execution of *kmeans*. During its compilation, the variables *j* and *nfeatures* are recognized as transaction local and barrier-free separately, and therefore the related runtime invocations are eliminated. According to the whole program, *feature* array is read only and doesn't

need to be synchronized either. However, the compiler is unable to recognize it, and since it is a pointer, the compiler performs full instrumentation to it, which is fatal to the overall performance.

To deal with this problem, the programmer could make use of a simple *private(feature)* clause as in Figure7. It enables the elimination of instrumentations to *feature*, which makes the program perform almost the same with the manually instrumented one, as illustrated in Figure6.

5. Conclusion and Future Work

In this paper, we present the design and implementation of Transactional Memory extension of the Open64 compiler, which supports replaceable underlying runtime library and several optimizations. We have performed a preliminary experiment, which shows that it is possible for the TM program compiled by our compiler to have a similar scalability with the original one with fine-grained locks.

Due to the time limit, only two applications are used in our tests. We are planning to perform more extensive experiments to verify our conclusions in the future. Regarding the design and implementation of the compiler, besides the problems that are discussed in each section, there are also other open issues. For example, we would like to provide mechanisms to allow programmers to register handlers for certain events such as abortion, function call, etc., in order to facilitate the programmers, and also offer opportunities to deal with I/Os in transactions. It is also critical to appropriately handle the conditional wait semantics. For instance, a literal analysis shows that 20 out of 55 critical regions in the whole PARSEC suites need signal processing. It is also very common to have signals within critical regions in commercial softwares. For example, the number is 28 out of 75 for FastDB. In addition, handling legacy code in the binary form in transactions, and whether supporting open nested or close nested transactions, are also open issues.

References

1. Justin E. Gottschlich and Daniel A. Connors, “Optimizing Consistency Checking for Memory-Intensive Transactions”, Proceedings of the 2008 ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)

2. Pascal Felber, Christof Fetzer, and Torvald Riegel, “Dynamic Performance Tuning of Word-Based Software Transactional Memory”, Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 2008

3. Dave Dice, Ori Shalev, and Nir Shavit, “Transactional Locking II”, In Proceedings of the 20th Intl. Symposium on Distributed Computing, 2006

4. Milos Milovanovoc, et al. “Transactional Memory and OpenMP”, In proceedings of the International Workshop on OpenMP (IWOMP), 2007

5. Intel C++ STM compiler prototype edition 3.0, http://software.intel.com, last modified on Dec 24, 2008

6. Woongki Baek, et al. “The OpenTM Transactional Application Programming Interface”, In proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), September 2007.

7. Pascal Felber, et al. Transactifying applications using an open compiler framework, The Second ACM SIGPLAN Workshop on Transactional Computing (TRANSACT07), August, 2007

8. Cheng Wang, et al. “Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language”, In proceedings of 2007 International Symposium on Code Generation and Optimization

9. Yang Ni, et al. “Design and Implementation of Transactional Constructs for C/C++”, In proceedings of The International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA), 2009

10. Martin Schindewolf, et al. “Towards Transactional Memory Support for GCC”, First International Workshop on GCC Research Opportunities, 2009

11. Tim Harris, Simon Marlow, Simon Peyton-Jones, Mauice Herlihy, “Composable Memory Transactions”, In Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, 2005

12. Torvald Riegel and Diogo Becker de Brum, “Making Object-Based STM Practical in Unmanaged Environments”, The Third ACM SIGPLAN Workshop on Transactional Computing (TRANSACT08), 2008

13. Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh and Kai Li, “The PARSEC Benchmark Suite: Characterization and Architectural Implications”, In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, October 2008.

14. Marek Olszewski, Jeremy Cutler, and J. Gregory Steffan, “JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory”, In Proceedings of The International Conference on Parallel Architectures and Compilation Techniques (PACT), September, 2007.

15. Chi Cao Minh, et al. “STAMP: Stanford Transactional Applications for Multi-Processing”, In Proceedings of The IEEE International Symposium on Workload Characterization (IISWC), Sept. 2008.