Reducing Logging Overhead for Deterministic Execution

Madan Das Gabriel Southern Jose Renau

Dept. of Computer Engineering, University of California Santa Cruz
{madandas, gsouther, renau}@soe.ucsc.edu
http://masc.soe.ucsc.edu

ABSTRACT

Deterministic execution of parallel applications can be enforced by logging speculative memory accesses and restoring saved state in the event of a conflict. However, logging all accesses has a prohibitive overhead. We propose three techniques to reduce logging in an always-on transactional memory system. The first, *Threaded Section Analysis*, statically determines when only a single thread is running, so instrumentation is unnecessary. The second, *Loop Invariant Log Motion*, reduces logging requirements for loops by moving logging operations outside loops. The third, *Multi-Threaded Read Only Memory*, reduces instrumentation significantly using simple programmer inserted directives. When combined, these techniques reduced logging requirements by an average of 63% in the benchmarks we evaluted.

1. INTRODUCTION

Parallel programming is significantly more difficult than sequential programming because the programmer must explicitly consider how the different parts of a parallel program communicate with each other. Threads have proved to be popular in shared-memory systems, allowing programs to communicate simply by sharing objects or pointers between threads. However, this simple communication mechanism creates a potential for errors such as deadlock, atomicity violations, and order violations. The difficulty is compounded by non-deterministic interleavings of memory accesses that vary from run to run, producing different results for the same input.

Recent research has proposed ways to enforce deterministic execution of multithreaded applications. By dividing execution into parallel and serial phases researchers have been able to take advantage of multiple processing cores to speed up application performance while maintaining the desirable properties of deterministic execution. However, deterministic execution systems incur overhead compared to non-deterministic systems, due to operations to synchronize accesses to shared memory, as well as thread serialization.

Bergan et al. [1] proposed CoreDet, a software framework for enforcing deterministic execution, which instruments loads and stores to enforce determinism. As a software-only framework, CoreDet implementation attains reasonable performance in part because of the observation that not all loads and stores need to be instrumented — only those involved with inter-thread communication. While this exact subset cannot be determined statically, many of the non-communicating loads and stores can be identified using static analysis techniques, and do not need to be instrumented.

In this paper we present three novel techniques to reduce instrumentation for software-only deterministic execution runtime systems. We do so by performing novel thread escape analysis which identifies memory accesses which are guaranteed to be data-race free, and consequently don't need to be instrumented. The proposed techniques can also be applied to logging based software transactional memory systems (STMs) or to data race detection mechanisms. To evaluate its effectiveness, we have an infrastructure similar to CoreDet as the baseline, and we add our three new optimizations to further reduce instrumentation overhead.

First, we propose *Threaded Section Analysis* (TSA) to statically analyze a program to identify its single threaded and multithreaded sections of execution. Only the multithreaded sections of code need to be instrumented to enforce deterministic execution. Using TSA we skip instrumentation of sections of an application that are guaranteed to execute only in single-threaded mode. A mod-ref analysis [2] is integrated with TSA to detect read only accesses during threaded sections.

Second, we propose *Multi-Threaded Read-Only Memory* (MTROM) which allows memory to be written to in single-threaded mode, but is read-only when the program is running in multithreaded mode. TSA with mod-ref is able to detect this pattern too, but due to imperfect alias analysis several read-only accesses can not be eliminated. MTROM marks a memory region during allocation as read-only during multithreaded execution. This explicit marking is propagated through the alias analysis to further reduce instrumentation. Since the programmer can incorrectly mark a region as read-only, we protect these pages as read-only during multithreaded phases to guarantee that the memory is not written and that the developer did not introduce any bugs.

Third, we propose *Loop Invariant Log Motion* (LILM) to separate the access instrumentation call from actual load or store. Similar to loop invariant code motion, we can then move the logging operation out of the loop body to its pre-header.

The rest of this paper is organized as follows: in Section 2 we describe the implementation of Threaded Section Analysis; in Section 3 we describe Multithreaded Read-Only Memory; in Section 4 we describe Loop Invariant Log Motion; in Section 5 we evaluate our experimental setup and results; in Section 6 we survey related work; and finally Section 7 concludes.

2. THREADED SECTION ANALYSIS

Threaded Section Analysis (TSA) is based on the observation that applications frequently are divided into parallel and serial sections. Using TSA we divide the program in disjoint thread sections or code regions that cannot execute simultaneously (Section 2.1). The advantage is that this enables improving thread escape analysis and detecting unnecessary accesses (Section 2.2). For example, the initialization code region may be done by a single thread which by definition cannot have sharing. Even more frequently, we can use a mod-ref analysis to detect code regions that have read-only accesses.

2.1 Threaded Section Detection

We define a threaded section as a program segment that completely encapsulates creation and joining (or cancellation) of threads that execute in parallel. The code segments immediately adjacent to a threaded section have only one active thread for the user program.

Definition: A *Threaded Section* (TS), represented by a tuple (S, T), where S and T are instructions, is a set of all basic blocks and instructions between S and T, both inclusive, such that all thread creations and completions in this region are dominated by S, and T is a post-dominator of all the instructions in this set, and the number of active user threads immediately preceding S and immediately succeeding T is 1.

We can think of a TS as the top level code for a parallelized part of a program. Since the threads created within a TS all terminate within the same TS, when we are concerned about synchronization operations between any two threads created in this section, we only need to consider code that is reachable from this TS. More precisely, it is the set of all functions that act as start functions for the threads, and the code in the parent thread that is executed in this section. Often, the parent thread just waits for the completion of child threads or acts as a master synchronizer.

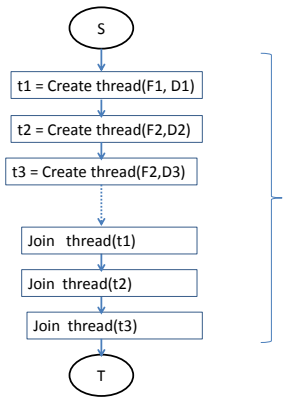

Figure 1: A typical threaded section of program

Figure 1 shows a typical threaded section, where three child threads are started, with function and data pairs of $(F1, D1)$, $(F2, D2)$ and $(F2, D3)$. Thus, a thread start can be seen as a tuple (F_i, D_i) where it executes function F_i on data set D_i. In the example shown, S is a dominator of the thread creations, and T is a post-dominator of all thread terminations. So, this TS can be represented as (S, T). Memory instrumentations can be limited to the threaded sections of a program. This is useful since in many complex programs, the portion of program that is performance-critical may be small, yet consume most of the runtime.

2.1.1 Identifying Threaded Sections

TSA is an inter-procedural analysis, so it is applied to the whole program. TSA uses a top-down recursive method to descend into the main function of a program, and looks for functions that create and join (or cancel) threads. If during a depth-first traversal of the function call tree, it finds a function that has both thread create calls and thread join calls, it considers that function for TS. If it detects that the post-dominator of all create calls dominates the dominator of all join calls, it considers this as a single TS. Otherwise, it finds subsets of thread create and thread join calls with the same relationship, in order to identify multiple TS in that code segment. In such cases, there will be multiple subsets of thread creations and thread joining, such that each subset holds the TS property described earlier.

TSA assumes that a function spawning threads also terminates or joins them. Note that only the top level thread sections are of interest to TSA, as outside those regions the program is running in single threaded mode.

After identifying a TS, TSA begins analyzing the thread start functions in each TS separately, in conjunction with the code in the root thread in the same TS. Since our focus is for the multi-threaded sections only, we can safely ignore instrumenting all functions that are not reachable from the thread start functions or from the main thread in threaded sections. The functions reachable from these regions are identified by a depth-first traversal of instructions in a TS.

2.1.2 Phase Analysis in TSA

Now, we describe how we extend TSA when each thread is comprised of multiple phases. This is a finer grained version of what we described in Section 2.1.1. In this case, we want to identify smaller tasks that can execute in parallel within each TS. A task boundary may be identified in different ways, mainly through a pthread_barrier call. Identifying the phases of a parallel section can improve the precision of analysis for the parts of the program that execute in parallel.

To decompose a TS, we look into the thread start functions and identify thread synchronization points in those functions. To simplify phase boundaries, we introduced a new directive, *sync_threads*, that acts as a sync-all barrier for all active threads. If each of the thread start functions in a TS ts contains exactly the same number of such synchronization points $(s_1, s_2, ...s_n)$, and s_{i-1} dominates s_i, then we have identified multiple phases in ts.

Figure 2 shows an example of phases in a threaded section that we are trying to identify with our algorithm. The parent thread creates p child threads in this threaded section. Each of the child threads is further decomposed into smaller phases $T(1, 1), T(1, 2)$ and so on at thread barriers. The entire TS begins at basic block B and terminates at T, where B to T is a smaller flow graph in the program CFG.

Since all threads arrive at *sync_threads* before any can exit that point, we can segregate memory accesses into disjoint sets corresponding to each of the phases. Hence, when we instrument memory accesses in each of these, we only

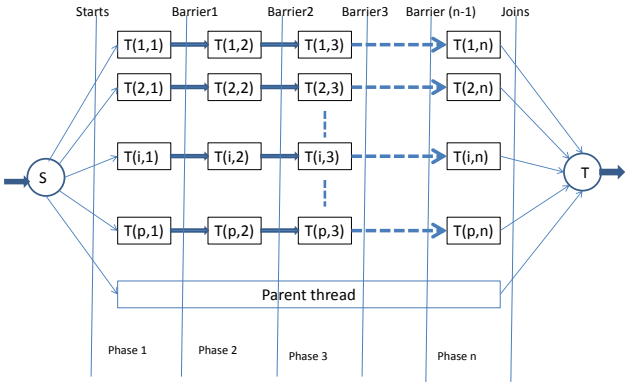

Figure 2: A decomposition of threaded section into smaller, disjoint parallel phases

need to consider the memory accesses within the context of that specific phase in conjunction with the parent thread. Since the parent thread is mainly waiting for other threads, we consider it to be overlapping with all phases in its entirety. Thus, if the program obeys some basic structure of multi-threading, the compiler is able to significantly improve synchronization overheads by eliminating much of the provably redundant access tracking.

To use multiple phases in a section, we do not require that all the thread start functions in that TS be the same. Rather, we only require that all of them have the same numbe of phases. This restriction might be relaxed, but we don't think that it is practical to do so. Also, it is not always possible to decompose these thread start functions into phases. We have incorporated two common situations where we think such decompopsition is practical to do:

– Straight Line Threaded Function: In this case, the global synchronization points are not part of any loop. There is a simple dominance and post-dominance relatioship from the function entry point to the first, second and last synchronization point, to the function return. Fig 3 (a) depicts the control flow of such a function.

– Looped Threaded Function: In this case, all global synchronization points are enclosed in a top level loop in the threaded function. The loop has a single exit point, which is the first branch. So, either all the synchronization points are exercised, or none of them are exercised. The global synchronization points still dominate and post-dominate each other. Special handling can be done for treating the program exits, such as the ones resulting from `assert` or `throw` calls within the phased code. If we ignore those outbound edges, then the CFG of such a threaded function should look as in Fig 3 (b). However, in this case, we have to merge the load/stores in the first part of the function with the last load/store set, as they may be executed in overlapping manner by different threads. It is okay for a particular thread to execute this outer loop fewer times than some other threads, since not executing the loop by a thread does not violate the condition that multiple threads are executing different sets of loads/stores. Each thread either executes only one set of load/store instruction in a particular phase, or they do not execute anything. Therefore, this segmentation of loads and stores into disjoint subsets is a sound method. Also, if a subset of active threads would return from the function earlier than others, the global barrier *sync_thread* would continue to work for other thread by its definition. In practice though, we expect that most often, all these threads to execute the loop the same number of times.

We also note than in the worst case, all the phases are collapsed into a single phase, implying that all loads and stores in the TS can potentially overlap with each other. The compiler pass marks the load and store instructions with the set of phases from which they can be invoked.

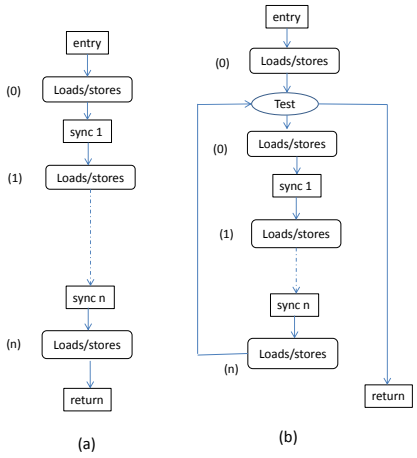

Figure 3: Structure of thread start functions suitable for phased thread section analysis. (a) Straight Line (b) Looped

2.2 Reduce Instrumentation in TS

The previous section show how to divide the program execution in different Thread Section (TS). Since the TS are disjoint, we just need to perform thread escape for each TS independently.

Data races are either of RAW, WAW and WAR, and require a write. If a TS section is executed by a single thread like in typical initialization, we do not need to instrument the memory accesses because we guarantee by definition that it cannot escape. Similarly, if a multithreaded thread section only performs read-only accesses in a given array, we do not need to instrument these loads.

While performing TSA, a set of points-to memory regions that are modified in a phase of a TS is created for each phase. Once phase-specific mod-ref information is available, the algorithm to instrument memory accesses checks if there is an overlap between the set of parallel phases executing that instruction and the set of phases writing to that points-to set. If there is no overlap between these two sets of phases, the memory access does not need to be instrumented.

TSA is based on a context insensitive, flow insensitive unification based pointer analysis method as described in [3].

Only its mod-ref analysis is context sensitive. Therefore, when it considers the possible set of pointers a particular instruction points to, it includes all possible points-to locations. In this scheme, each array is treated as a different memory region. In the absence of array bound reads and writes, and unknown function calls within the thread sections, this conservatively considers all possible aliasing. For unknown function calls, it conservatively assumes that such a function can modify any memory location. TSA is interprocedural and is applied on the whole program. So, it is aware of all the functions that are called at any place within each TS.

2.3 Limitations of TSA

TSA currently only identifies TS within the same function. This seems to be the most common case, and is not a critical issue. If the thread creations and joining happen in different functions, then the programmer can most likely move them into a single function.

TSA relies heavily on the CFG of the program, especially in the region of interest for parallel segments. Sometimes, if there are complex branching or exits (for example, due to asserts), which break the dominance and post-dominance relationships of start and end points of a thread section, TSA fails to identify the TS. In such cases we fall back on a less precise method, where we simply treat all thread creations as being in the same TS, along with the whole function body that starts and ends all the threads. This secondary approach also provides a significant improvement in mod-ref analysis, though not as precise as desired. We also experimented with user directives and special primitive functions that can be used by the programmer to demarcate the TS and synchronization points within a TS. These directives help the compiler optimize analysis of those regions, assuming the user-inserted directives are correct.

3. MT READ ONLY MEMORY

In this section, we describe Multithreaded Read-Only Memory (MTROM), that we define as a region of memory that is read-only, only when an application has multiple active threads. For data races to happen across multiple threads, at least one thread accessing a shared memory location must perform a write to that location while some other thread reads or writes to the same location in the same thread section. It is quite common practice to have a single thread write to memory locations during an initialization phase, and during the parallel phase of the program, multiple threads only perform reads of that memory region. A very common example of this is the parallel dense matrix multiplication. During the parallel phase of the program, the matrices being multiplied are not written to, only the product matrix is modified. Hence, the input matrices can be allocated from MTROM memory space. Another example of the usefulness of this technique is in the case of large graphs where additions and deletions only happen in single-threaded mode, but the query operations happen in multi-threaded mode. In the benchmarks we tried, many exhibited this nature of parallel execution.

Ideally, static analysis as described in Section 2.2 would identify all such accesses. But due to limitations in pointer analysis and other static analysis methods in tracking large number of objects allocated at different places, it is not always possible to do so. In the MTROM approach, the programmer provides a hint to compiler by allocating heap memory from MTROM regions that is not supposed to be written in multi-threaded mode. The programmer can also mark some globals as MTROM, if desired. In practice, we observed that most large chunks of such memory are heap allocated or mmaped, as their sizes are not known upfront. We provide MTROM versions of functions (such as `malloc` and `mmap`), and require the programmer to use them when allocating MTROM memory. The pointer analysis then marks the memory objects created by these methods as 'MTROM heap', and similarly for global memories, it marks them as 'MTROM global'. During access instrumentation in thread sections, if a read memory access is traced to a points-to set that points to only MTROM-heap or MTROM-global regions, then that memory read access is not instrumented. On the other hand, if there is a write access to a points-to set containing only MTROM objects in some parallel section of the program, then the compiler pass can flag that as an error or warning. If the points-to set is a mixture of both MTROM and non-MTROM memory regions, then the read and write operations are instrumented as usual. The runtime library protects MTROM regions from writes during parallel phases by marking the pages read only.

To detect incorrectly marked MTROM memory regions, a test can be be implemented either in all the executions or during debug mode. To avoid adding overhead with additional checks in the write logging functions, we propose to protect as read-only the memory pages marked as MTROM. The read-only protection happens during multithreaded execution, and they are marked as read-write whenever the program goes to a single thread execution phases. These checks are to detect incorrect MTROM annotation by the programmer, and also to detect inadvertent writes to such memory locations in parallel mode.

In the benchmarks we analyzed, using MTROM required minimal changes (a few lines) to the source code. The only change was to replace few `malloc`, `mmap`, `free` and `munmap` with their mtrom equivalents, and annotating at one place to ensure that the memory being accessed is mtrom. This method is not automatic, but it is robust and general, since the writes to such memory are protected in parallel phases. It provides a strict guarantee to the programmer that there are no races through the MTROM memory regions.

4. LOOP INVARIANT LOG MOTION (LILM)

We describe a novel technique to reduce overhead of logging array accesses in loops. We observe that if the address of an array does not change, then we can make a single call to log all the accesses to its elements in one call to the runtime library. Depending on the size of the array element, this can significantly reduce redundant calls. This is applicable to both read and write accesses. The loop must not be split across multiple tasks (or quanta) for this optimization to work.

Example 1 A simple loop with loads and stores

```
for(int i = start; i < end; i++) {
   C[i] = A[i] * B[i];
}
```

In Example 1, the read and write operations are not loop invariant, and hence cannot be hoisted outside the loop.

However, for logging, we are only concerned with the range of addresses accessed in the loop, and not the actual data content. So, instead of replacing each read and write with a call to the runtime library, which can be expensive, thus calling the logging routine $end - start$ times, we can make a single call to a logging function for the whole range (start, end) for arrays A, B, and C. The logging function is optimized to stride through the array, as element size is known. Assuming that the array element size is S, the cache line size used by the runtime model is C, and the number of loop iterations is N, this will call the logging function $\lceil N * C/S \rceil$ times. Without this optimization, the logging function is called N times. If S is a 32-bit integer or float, and C is 32 bytes, this reduces the number of calls eight fold.

This optimization is most useful when we have unit step functions for the loop induction variable. With a small stride step function, this may still provide some benefit. A unit step seems to be the most common case in practice.

However, we have only been able to apply this optimization to the last dimension of a multi-dimensional array. For example, in nested loops for matrix multiplication, the pass is only able to optimize the accesses to the elements when consecutive elements are accessed in successive loop iterations. Although one theoretically could hoist an optimal number of logging operations outside of nested loops, we haven't investigated this yet. It is an item we plan to explore in the future.

5. EVALUATION

We implemented TSA as a compiler pass using the LLVM compiler framework [4]. Part of our analysis relies on Data Structure Analysis (DSA) [3]. DSA is a static pointer analysis, and provides the flow insensitive pointer analysis framework for this research. DSA already identifies some memory locations as thread local, and hence non-escaping.

We are developing an always-on STM runtime library that enforces deterministic execution and we used this for our evaluation. For the compiler pass and optimizations, we used LLVM 3.1. We incorporated CoreDet-like optimizations including thread escape analysis, successive access optimization, and coalescing nearby accesses into single logging operations.

We evaluated the effectiveness of our techniques as a percentage reduction in memory accesses for instrumented code. The main reason to choose a percentage scale for these data is that the absolute numbers have a wide variation across these benchmarks, as they depend on input size and program type. A naive instrumentation will instrument 100% of the memory accesses. The various columns of the table show the percentage of these accesses reduced when a combination of techniques is applied. The reduction is thus a cumulative effect of applying all the techniques. The table entries show the percentage of accesses that did not need instrumentation due to the applied methods. The following abbreviations are used for the column headings:

– opt1: CoreDet-like optimizations

– opt2: opt1 + TSA single thread accesses

– opt3: opt1 + TSA read-only accesses

– opt4: opt3 + Loop Invariant Log Motion (LILM)

– opt5: opt4 + MTROM

Benchmark	**opt1**	**opt2**	**opt3**	**opt4**	**opt5**
blackscholes	<1	<1	92.5	92.5	92.5
fft	34.8	34.8	52.4	52.4	52.4
lu	32.7	33.3	33.3	33.3	33.3
radix	30.1	40.0	44.9	63.5	63.5
histogram	33.7	33.7	66.8	66.8	66.8
kmeans	<1	1.0	50.6	75.2	75.2
matrix_multiply	<1	<1	<1	<1	51.1
pca	<1	<1	99.8	99.8	99.8
reverse_index	<1	<1	<1	<1	24.5
string_match	23.2	23.2	58.6	58.6	58.6
linear_regression	60.7	60.7	62.0	62.0	62.0
Average	**19.8**	**20.8**	**51.1**	**55.0**	**61.8**

Table 1: Total access logging reduction as percent of naive instrumentation.

Table 1 shows the reduction in the number of instrumentation calls to the runtime library with the various optimization techniques as a percentage of naive instrumentation counts. The first column shows the reduction using optimization techniques similar to CoreDet. In some cases, TSA is able to identify the memory points-to set properly.

Benchmark	**opt1**	**opt2**	**opt3**	**opt4**	**opt5**
blackscholes	<1	<1	100.0	100.0	100.0
fft	<1	<1	30.9	30.9	30.9
lu	<1	<1	<1	<1	<1
radix	15.7	30.9	38.5	61.4	61.4
histogram	<1	<1	50.3	50.3	50.3
kmeans	<1	<1	50.4	75.1	75.1
matrix_multiply	<1	<1	<1	<1	51.2
pca	<1	<1	99.9	99.9	99.9
reverse_index	<1	<1	<1	<1	39.3
string_match	36.9	36.9	100.0	100.0	100.0
linear_regression	87.3	88.2	100.0	100.0	100.0
Average	**13.1**	**14.5**	**52.0**	**56.3**	**64.4**

Table 2: Load logging reduction as percent of naive instrumentation.

Table 2 shows the reduction in dynamic load instrumentation counts when using the various optimization techniques as a percentage of naive instrumentation counts.

Our runtime library is still under development; however, we have promising results for the instrumentation techniques described in this paper. We included results from the benchmarks from the PARSEC, SPLASH, and Phoenix suites that are currently able to compile and run: blackscholes from PARSEC; fft, lu, and radix from SPLASH; and pca, histogram, kmeans, linear regression, matrix multiply, histogram, kmeans, matrix_multiply, pca, reverse_index, string_match, linear_regression from the Phoenix benchmark suite. Our observations on the benchmarks evaluated are as follows:

– blackscholes: blackscholes iterates over 4 global read vectors. TSA is able to detect that most of the accesses are read-only (opt3), and reduces load access logging by 100%. For store accesses, the same array is written by multiple threads. Without a sophisticated range analysis to partition the array statically, we cannot eliminate the store logging operations. But overall, we still achieve a 92.5% logging reduction.

– FFT: In FFT, there is a global array of input time domain points that is not modified. As the input point set is global and is not modified, TSA can detect this read-only accesses, and reduce load counts by 30%.

– lu: In LU, threads modify the input matrix in-place with complex strides. Our methods do not show significant improvement, except for reducing some access tracking that happens in single threaded mode. LU performs significant number of accesses with a daxpy method. Our LILM does not correctly detect this as a vector access. If we avoid inlining the daxpy function over half of the memory accesses can be removed. Nevertheless, we did not do this optimization because we expect to improve the vector detection mechanism.

– radix: radix performs in-place sorting of the keys. We see some benefit from avoid instrumentation during initialization (opt2), and little benefit from detection read-only accesses (opt1 or opt5) or MTROM based analysis. LILM is the most effective technique with 19% reduction in instrumented loads. All the techniques combined reduce the access logging by a cumulative 63%.

– histogram: The threads read values from global data, and increment counters each time they read a value. Hence, we see a reduction of 50% in load count, and 33% in total access count when TSA detects read-only accesses (opt3). Coalescing of nearby memory accesses helps this case quite a bit, so opt1 reduces 33% of memory accesses. Also, as almost all writes are to thread-local variables, gives a very good savings in number of store logging.

– kmeans: kmeans has a very high load-to-store ratio, stores being only a small fraction of total accesses. It also uses a global array of points, so TSA can determine the several read-only accesses Also, LILM is able to move some of the loads into the preheaders of two loops, resulting in an eight-fold logging efficiency improvement, for a 32 byte cache line size and 4 byte element size. Overall, access count is reduced by 75%.

– pca This program has two separate threaded regions. In the mean computation region, it is reading matrix data, and writing only the means of each row. In the subsequent threaded region to compute covariance, mean is only read. Hence, in the second threaded section, both the matrix data and mean data can be treated as read-only and do not need instrumentation.

– matrix multiply: This is a classic case of MTROM usage (opt5), where the input matrices are not modified at all during the multiplication process. So, the MTROM method is able to eliminate some of the loads from the MTROM memory. Due to weaknesses in pointer analysis, all accesses could not be traced to mtrom-only heaps. We are investigating this currently for possible improvements.

– reverse index: A write to null-terminate the end of the link in a large data set renders that memory read-write, although it is really MTROM in principle. We modified the code slightly to avoid this, and carried the length of the link in the data separately, rather than having a null-terminated link. This optimization reduced the memory tracking need for this benchmark by 34% with MTROM (opt5). Without this modification, MTROM did not produce significant benefit for this case.

– string match: The baseline CoreDet techniques (opt1) already optimizes 25% of the accesses. TSA with read-only (opt3) helps reduce more load accesses. MTROM makes the load tracking almost unnecessary. Due to a weakness in the static pointer analysis, we believe the store accesses weren't eliminated significantly, and might be improved.

– linear regression: In this case, there are successive loads of the fields of the same structure. So, CoreDet-like optimizations could reduce load tracking significantly (opt1). TSA is able to detect all the reads as read-only (opt4) so opt4 shows 100% load reduction, with an overall 62% logging reduction. There is a significant amount of stores in this case, which results in smaller reduction in total access compared to load logging reduction percentage.

With an ideal pointer analysis, TSA with mod-ref analysis should make the use of MTROM unnecessary. In the evaluated benchmarks only matrix multiply and reverse index benefited from MTROM (opt5). It could have been possible to further reduce the instrumentation in some other benchmarks but it may have required changing the code.

As expected, avoiding code initialization (opt2) reduces overhead, but not significantly. Opt2 avoids code instrumenting code initialization, but also phases of the execution with single threaded mode. Nevertheless, in the evaluated applications it did not happen.

A significant instrumentation reduction is achieved when TSA is combined with mod-ref analysis to detect read-only accesses. This technique (opt3) avoids instrumenting over 37% of all the loads.

In some cases, a true data partitioning is needed within the same contiguous memory segment, such as in parallel sorting of data as in radix, or modification of common data as in lu-factorization. These are harder to detect statically with TSA, and need complex solvers to prove that they are non-overlapping accesses.

6. RELATED WORK

Recently several proposals have been made to provide deterministic execution for traditional multithreaded C and C++ programs. CoreDet [1] is a compiler and runtime environment that together enforces deterministic execution for multithreaded programs even under race conditions among various threads. CoreDet decomposes the instructions of threads into manageably sized blocks by instruction count, which it defines as quanta. It uses multiple techniques to achieve such determinism. CoreDet also uses some compiler techniques to reduce its logging overhead, and our framework includes techniques used by CoreDet. CoreDet also uses the LLVM compiler [4], like our work. The optimizations that we have proposed in this paper are complementary to work done for CoreDet and could potentially improve its performance.

DThreads [5] is another recent work that provides a deterministic execution model. In DThreads, threads are implemented as processes and OS memory protection enforces isolation between threads. Writes to shared memory are protected, and local copies of entire pages are made on write, and maintained until the next commit point. DThreads doesn't rely on memory access tracking for each access. It

also avoids instrumenting single threaded code, as in our TSA, by switching between execution modes at runtime. However, it relies on OS-controlled virtual memory protection, and switching execution modes at runtime is more difficult for systems that instrument memory accesses.

Our analysis framework is not just aimed at providing a deterministic runtime model, but to aid all such systems focusing on race detection and debug, through rigorous compiler support for multi-threaded programs. It works for general shared memory multi-threading models, so program memory is not duplicated for each thread. It is also very useful for log based STMs. In [6], authors argue that STM overhead is so high that it is merely a research toy. With our proposed techniques, we believe STMs can become significantly more efficient, to make them almost practical.

There has been previous work to reduce STM overhead, mainly for strongly typed languages. For example, in [7], the authors describe object ownership in Java to reduce logging operations in STM. [8] also describe escape analysis for object oriented languages like Java. In [9], authors describe something similar, called *May Happen In Parallel*, that works with java synchronizations. Our analysis is more rigorous, and is context sensitive to one specific threaded section. Even if two functions may both be called in parallel, there is no overlap if they are not in the same TS. Our analysis considers the actual memory regions accessed in each parallel phase separately. In [10], the authors describe somewhat similar efforts directed at reducing STM overhead in the compiler. Our methods are different by nature. While their method focuses on some optimization specific to code motion and read-only transactions, we developed a framework for rigorous analysis of the threaded regions for C and C-like programs.

Many researchers have focused on improving the precision of pointer analysis for threads. We observe that it is equally important to consider the modified and referenced properties of the pointer values. A major theme of our proposal is to use the precision of points-to relationship in conjunction with modified-referenced information for each parallelized segment of a program to reduce memory access tracking requirements.

7. CONCLUSION AND FUTURE WORK

We have presented three techniques: threaded section analysis, loop invariant log motion, and multithreaded read-only memory, each of which can be used to reduce instrumentation overhead for deterministic execution systems. Each technique can be used separately; however, when combined they reduce instrumentation overhead by an average of 63% for the benchmarks that we evaluated.

Although we have focused our work on reducing instrumentation overhead for deterministic execution systems, the techniques we have described can be useful for reducing overheads in software transactional memory or in data-race detection. Any instrumentation technique requiring instrumentation when running in multithreaded mode but not single-threaded mode could benefit from our proposal.

Deterministic execution has the potential to simplify parallel programming, but current proposals (particularly software implementations) have significant performance overheads. Reducing instrumentation can reduce this overhead and improve performance, and the techniques we have presented are a valuable addition to achieving this goal.

8. REFERENCES

[1] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman, "Coredet: a compiler and runtime system for deterministic multithreaded execution," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, New York, NY, USA, 2010, ASPLOS '10, pp. 53–64, ACM.

[2] William Landi, Barbara G. Ryder, and Sean Zhang, "Interprocedural modification side effect analysis with pointer aliasing," in *In Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, 1993, pp. 56–67.

[3] Chris Lattner and Vikram Adve, "Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, Chigago, Illinois, June 2005.

[4] http://www.llvm.org, "Low level virtual machinepointer analysis – a survey," .

[5] Tongping Liu, Charlie Curtsinger, and Emery D. Berger, "Dthreads: efficient deterministic multithreading," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2011, SOSP '11, pp. 327–336, ACM.

[6] Calin Cascaval et. al., "Software Transactional Memory: Why Is It Only a Research Toy?," in *ACM Queue - The Concurrency Problem, Volume 6 Issue 5*, September 1 2008, pp. 46–58.

[7] Nels E. Beckman, Yoon Phil Kim, Jonathan Aldrich, and Sven Stork, "Reducing stm overhead with access permissions," .

[8] B Blanchet, "Escape analysis for object-oriented languages: Application to Java," in *In Proc. ACM SIGPLAN Conf.OOPSLA ÂŠ99.ACM SIGPLAN.* 1999, ACM Press.

[9] Clark Verbrugge Lin Li, "A Practical MHP Information Analysis for Concurrent Java Programs," in *Lecture Notes in Computer Science, Languages and Compilers for High Performance Computing*, 2005, vol. 3602, pp. 194–208.

[10] Arie Zilberstein Yehuda Afek, Guy Korland, "Lowering STM Overhead with Static Analysis," in *The 23rd International Workshop on Languages and Compilers for Parallel Computing Proceedings Springer-Verlag (LNCS), Rice University, Houston, Texas, USA*, October 7 - 9 2010.