1 Introduction

The number of cores on a CPU chip is currently doubling every two years, in a manner consistent with Moore’s Law. This is leading to an era of many-core computing [1, 2, 3], in which thread parallelism is desirable for the HPC community for the following reasons. First, those sequential algorithms, which were previously hard to parallelize on distributed-memory computer clusters, can sometimes be made thread-parallel and run on a many-core shared-memory computer. Second, many traditional HPC applications that have been developed for computer clusters, can sometimes be adapted to run more efficiently with many threads [4]. Under such circumstance, the HPC community is eager to address three important issues concerning the thread parallelism on many-core platforms:

1. Given a sequential program (or a parallel one on distributed memory), how does one efficiently transform the code to be multi-threaded?
2. How can one know if the resulting multi-threaded code is still correct?
3. How can one guarantee scalability by eliminating possible performance bottlenecks arising from shared memory?

Significant progress was obtained in solving those issues for three widely varying applications: Geant4 [5], ILU(k) [6] and AliRoot [7]. This research will review these case histories, and demonstrate a methodology for transforming sequential program into multi-threaded code. Out of these three case studies, we will draw general conclusions about the three questions above: the benefits and limitations of automating the process of code transformation; and the issue in verifying (both statically and dynamically) the correctness of the transformation; and efficiency bottlenecks (e.g., the frequent memory allocations by multiple threads common in C++).

Based on this experience, this work proposes a semi-automatic method: Scalable Task-Oriented Code Transformation (STOCT). STOCT adds thread parallelism to sequential code via a series of code transformation steps. STOCT’s goal is:
Given a C/C++ software program, transform the source code to replace several independent copies of a sequential process with an equivalent single process consisting of several threads, in order to take advantage of many-core computers in a memory-efficient and scalable manner.

The implementation of STOCT takes advantage of two critical techniques: (i) moving large parts of the data segment to the thread-local storage (TLS [8]) for thread safety, which later requires a non-standard extension for memory allocators; and (ii) protecting memory pages for write-access reflection, from which one derives a run-time tool for elimination of performance bottlenecks, for correctness verification and for data-race arbitration. The following paragraphs briefly describe the proposed STOCT methodology.

1.1 Transformation for Thread Safety (TTS)

The essential spirit of TTS is to make all global variables and static variables thread-private. For this purpose, TTS is developed by iterating two steps: collect information about all global variables and static variables; make each global or static variable thread-local. Because each target application may have its own idiosyncrasies, one must iterate in the development of TTS by observing what variables were not properly captured in the previous transformation. TTS does not terminate until all such variables are accounted for. The transformation TTS generates unconditionally thread-safe code in the sense that any thread can call any function and guarantee the absence of data races. By replacing processes with independent threads, the Inter-Process Communication (IPC) for synchronization of a formerly distributed computation is completely eliminated.

1.2 Transformation for Memory Footprint Reduction (TMR)

The transformation TMR modifies source code further to share relatively read-only data, which is defined as follows. A variable may have been written to during its initialization, but may be read-only thereafter, or more specifically, during the computation of each task. Such a variable is referred to as relatively read-only and is sharable among threads. TMR iterates three steps: (i) select some possible relatively read-only data structures and check whether they are written to during the execution of each task; (ii) separate non-relatively read-only fields (transitory data) from data structure and thread-localize them; and (iii) initialize relatively read-only data, and replicate the transitory data. TMR terminates when the memory footprint has been sufficiently reduced to satisfy the design goals.

1.3 Debugging Tools

TMR may introduce two kinds of errors: (i) an incorrect thread initialization due to a design fault in the third step above; and (ii) a data-race due to a false positive in the first step of the previous section. To debug the first kind of error, we use a bi-simulation tool to compare the original program with the multi-threaded version to verify whether they have an identical behavior when a computation is run for the same set of tasks. One runs both the original and the transformed code on the same set of tasks. The transformed code is run using a single thread. The original code may already be just one single-threaded process — or if it was a distributed parallel computation, it is run with just a single node.
This tool follows the same idea as the standard proof via bi-simulation in which the original program is regarded as the “specification” and the multi-threaded version is regarded as the “implementation” [9].

The second kind of error occurs only during the production-run phase. To capture these errors, the target computation is run in a traced mode under control of a separate coordinator process. To handle this category of error, memory pages for shared data are write-protected. Whenever this part of memory is updated, a segfault signal (segmentation fault) will be generated, which causes the coordinator process to schedule threads of the target application to sequentially pass through this unintended critical section of code. The memory protection can be used in an “always-on” mode to guarantee runtime correctness. For this purpose, runtime correctness is defined as follows:

*Unintended updates to shared data should be executed only by a single thread during that thread’s current task, and during this time, no other thread should change any shared data.*

1.4 Thread-Private Malloc Library (*TPMalloc*)

The object-oriented feature of C++ encapsulates memory allocations for the string class and object container templates, whose implementation requires the dynamic allocation and freeing (malloc/free) of memory. As the number of threads grows, intensive concurrent mallocs/frees lead to a performance degradation, that has been intensively investigated by many developers seeking to develop “smarter” allocators such as tcmalloc [10], ptmalloc3 [11] and hoard [12]. Although these implementations alleviate the performance degradation when the number of threads is not large, e.g., less than 8, they never eliminate the performance degradation completely. This is because the malloc standard requires the use of a shared memory data structure so that any thread can free the memory allocated by any other thread.

We propose here TPMalloc, a non-standard extension to the standard malloc specification. TPMalloc produces a private heap for each thread in addition to the original central heap shared by all threads. In this thesis proposal, we are concerned with the case of each thread executing an independent task. This occurs because the transformed code arises from a previous sequential application. This situation encourages the use of an optimization in which each thread executes a task by bypassing the shared central heap and uses a thread-private allocation arena.

1.5 Avoidance of Cache Coherence Bottlenecks

To analyze the root cause why standard memory allocators suffer from performance degradations, we find a common problem of multi-threaded applications on many-core computers: writes to shared variables, even with a lock, generate excessive cache misses. This is because a write by one CPU core forces the chipset logic to invalidate the corresponding cache lines of the other cores. Even if a shared-variable write results in a cache hit for one thread, all other threads that include this shared variable in their active working set will eventually experience a cache miss (*coherence miss*).

The above problem is further exacerbated on motherboards with two or more CPU sockets. This is because such motherboards frequently omit fast off-chip caches shared among the CPU chips. In general, we claim:
Theorem 1.1. The coherence miss rate due to intensive updates to a shared variable will increase as applications use more and more CPU cores or CPU chips.

**Proof.** Suppose there are $w \geq 2$ thread workers that update the same variable. The total number of write accesses is designated as $n$. During the task-handling phase, write accesses are assumed to be evenly distributed. The optimal load-balancing for thread parallelism is also assumed, so that the number of updates is $n/w$ per thread. Consider any update access $u_1$ and its preceding access $u_0$ by the same thread. A cache miss results from $u_1$ whenever the variable is changed by another thread between the times of access for $u_0$ and $u_1$. It is easy to see that any write access from another thread falls between $u_0$ and $u_1$ with probability $w/n$. Therefore, the probability that no update happens during this period is $(1 - w/n)^{n - n/w} \approx e^{1-w}$. Furthermore, one cache miss happens at $u_1$ with probability $1 - e^{1-w}$. When $w$ is large, most of the $n$ write accesses will experience a cache miss.

**Example.** Consider a computer with four Intel Xeon X7460 CPUs (24 cores in total). Each core has its own L1 cache, and it suffices to set $w$ to be 24, the number of cores for analyzing this L1 case. There is also a single L3 cache per chip. Hence, for L3 cache, one can consider the threads of one chip as a single large super-thread and apply the formula by setting the number of threads $w$ to be the number of super-threads, or equivalently the number of chips. Hence, $w = 4$ in this L3 case. Similar ideas can be used for analyzing L2 cache with three L2 caches per CPU chip. For this case, a $w = 12$ analysis is appropriate.

These cache coherence issues are closely related to correctness. If read-write access to certain shared variables occurs outside of a lock, then the transformed code may be incorrect. One can fix such bugs by adding locks. But locks also slow down the program, while not addressing the underlying issue of loss of performance as the computer hardware tries to maintain cache coherence.

To avoid both locks and coherence misses, STOCT enforces the principle that in the transformed code,

*shared application data is initialized by the master thread only and is never changed by any task.*

This principle is a prerequisite for high scalability. It guarantees that the parallel computation for independent tasks is data-race free as well. As a result, locks are unnecessary.

### 1.6 Data-Race Coordinator (DRC)

The data-race coordinator (DRC) is a crucial tool required for three of the scenarios of this proposal: (i) given a class, recognize transitory member data, as defined in Section 1.2; (ii) for the performance bottleneck mentioned in Section 3.2.2, detect all shared variables that may incur write accesses; and (iii) during the production-run phase, guarantee the run-time correctness, as described in Section 1.3. The DRC tool proposed here is based on write-protection of RAM. It is a critical tool, since memory manipulation is the very essence of shared-memory computation. The incorrect ordering for memory update events is the core concern when dealing with data races and many other parallel programming bugs.
1.7 Structure of Proposal

The rest of this proposal is organized as follows. Section 2 reviews some related work. Section 3 examines the STOCT methodology and describes the STOCT design toward scalable task-oriented thread parallelism. Section 4 discusses some examples that demonstrate the capacity of STOCT. Finally, Section 5 presents a schedule of milestones, leading to the completion of the proposed dissertation work.

2 Related work

Parallelization is always a focus for the HPC community. Some representative approaches on programming, debugging and performance profiling are summarized in the following sections.

2.1 Compiler Based Parallelization

Started from early 1990s, compiler-based parallelization methodologies considered primarily Fortran programs because Fortran makes stronger guarantees about aliasing than C/C++. Some Fortran parallelization compiler includes: Rice Fortran D compiler [13], Vienna Fortran compiler [14], Paradigm compiler [15] and Polaris compiler [16]. This work originally targeted data parallelism for distributed-memory machines.

Another parallelization approach was SUIF [17] from the mid-1990s. This work introduces a parallelizing compiler based on an intermediate language for both Fortran and C programs. This parallelizer translates sequential programs into parallel code for machines with shared-address spaces and generates a single-program, multiple-data program that contains calls to a portable run-time library.

2.2 Parallelization for Particular Applications

Some recent multi-threading work for linear algebra algorithms is: PLASMA [4], which addresses the scheduling framework; and PLUTO [18] and its variant [19], which addresses the compiler-assisted parallelization, optimization and scheduling. This work considers C programs in which the compiler-based methodologies are suitable for data parallelism existing in tractable instances of nested loop.

The issue of multi-threaded access to shared variables has been an issue for wider use of multi-threading. There are still newer approaches that attempt to address this issue. For example, commutativity analysis [20] automatically extracts parallelism from utilities such as gzip, bzip2, cjpeg. Some researchers recommend to introduce nondeterminism only explicitly and sparingly, as discussed in [21]. In contrast, still other researchers present Kendo [22]: a software framework that enforces “weak determinism”. Sometimes the parallelization method is not automatic, as demonstrated by the parallelization for a web browser based on application anatomy [23].

2.3 Parallel Programming

OpenMP [24] directives and APIs support parallel programming through enhanced compilers to reduce the development manpower. Another approach is provided by Cilk [25]. Both approaches rely on a runtime libraries to schedule the parallel computation on the given platform. Automatic
introduction of OpenMP into sequential applications is a topic of current interest for compiler-based parallelization methods. A recent work, ROSE [26], extends this methodology by focusing on C++ STL containers and complex user-defined class types. It explores compiler techniques to recognize high-level abstractions and exploits their semantics for automatic parallelization.

The task-oriented parallel C/C++ (TOP-C [27]) describes a different approach from those mentioned above. The TOP-C philosophy is to parallelize sequential code, while reusing as much of the original sequential code as possible. The resulting parallel algorithm is derived by identifying tasks to be computed in parallel from the original sequential algorithm. Another goal of TOP-C is to provide a parallel algorithmic skeleton to unify the parallel programming for both distributed memory clusters and shared memory machines. Some sample parallel algorithms such as parallel GCD and parallel Gaussian Elimination are provided as examples in TOP-C. Since its emergence in mid-1990s, TOP-C has been primarily applied to software parallelization for distributed-memory [28, 29, 30, 31, 32, 33], although it has also been used for thread-parallelism [34, 35].

2.4 Debugging on Multiple Threads

It is widely acknowledged that concurrent programming is difficult [21] and error-prone. Data races are a particularly important problem. Since object-oriented programs associate data with the code that manipulates the data, it is relatively easy to place a lock around code manipulating the given data. But in general, data races are serious, and it is too easy to create new data races when updating older code. Some tools for data race detection are: SharC [36], which checks data sharing strategies for multi-threaded C via declaring a data-sharing strategy and run-time checking of correctness of the declared strategy; RELAY [37], which is used to detect data races in the Linux kernel; RacerX [38], which has found serious errors in the Linux and FreeBSD kernels; and KISS [39], which has obtained promising initial results in Windows device drivers. Although these methods have demonstrated their utility, they do not guarantee to be applicable for arbitrary programs. Hence, as programming transformations, they are unsound.

Two sound static data race detection tools are: LOCKSMITH [40], which finds several races in Linux device drivers; and the method of Henzinger et al. [41], which is based on model checking. Sound methods need to check a large state space and may fail to complete due to resource exhaustion. All these methods, even with their limitations, are crucial for system software (e.g., an O/S) which requires strict correctness because it is intended to run for long times.

With object-oriented programming, most data races are eliminated after accesses to shared data are encapsulated through careful design. Under such circumstance, most parallel programming bugs fall into the category of order-violation [42]. Similar bugs occur when developers cut large transactions into small pieces due to a performance consideration. The execution of threads can interleave in an unexpected order for the processing of those sub-transactions. For this kind of bug, developers have to check many execution branches in order to reproduce potential faults [43].

2.5 Performance Profiling

Execution profilers are crucial for programmers to find the root cause whenever performance bottleneck exists in parallelized programs. Some profilers such as GNU Gprof [44], Quartz [45], and OProfile [46] gather runtime statistics for later analysis. Gprof reports per function cost/times and a call graph to indicate caller/callee times. Quartz follows the same philosophy as Gprof: relating these performance metrics to one another and to the structure of the program. Nevertheless,
Quartz targets parallel programs and reports normalized processor times as a principal metric tied to the logical structure of the program. Similarly, OProfile can monitor hardware events and relate the number of cache misses to specific functions.

Some simulators such as MemSpy [47], CProf [48] and Valgrind’s CacheGrind module [49] help developers locate poor cache access patterns. A recent profiler, DProf [50], adopts a hybrid method to introduce a data profile, and reports the following information:

- the data types with the most cache misses; and
- a data flow graph to summarize
  1. how objects of a given type are accessed throughout their lifetime; and
  2. which accesses incur expensive cross-CPU cache loads.

Even with all those tools, it remains difficult for any general method to attribute performance drains to instructions at the source code level: insufficient instrumentation loses the information concerning the relationship between statistics and instructions; while too much instrumentation slows down threads, and leads to imprecise observations.

3 STOCT Methodology

STOCT leverages the TOP-C philosophy, and provides a feasible methodology to obtain a thread-safe yet scalable version from a legacy code base for large programs. The TOP-C philosophy is important for those programs, where any radical change is either infeasible or impractical.

STOCT addresses the TOP-C model to bridge the gap between sequential programming to concurrent programming. Along these lines, STOCT decomposes four intertwined design efforts: to exert more parallelism, to share more data, to guarantee correctness and to achieve scalability into separate software processes. More concretely, STOCT has the following features:

1. **Thread safety** is achieved by TTS for the maximal parallelism where only the minimal shared data set is allowed, that is constant variables and data arrays. TTS takes advantage of TLS, and is similar to well-known approaches such as the private data clause in OpenMP [24] and Cilk [25], and the SUIF [17] privatizable directive. Such directives can be issued either by the programmer or by the compiler. TTS targets large C/C++ software containing many virtual functions and callback functions — a context that would overwhelm both concurrent programming and compile-time analysis.

2. **Memory footprint** is reduced by TMR for threads to share more and more relatively read-only data and replicate less and less transitory data. TMR can be considered as a coding effort to reorganize data structure for data parallelism while maintaining thread-safety.

3. **Transformation correctness** is relaxed to a runtime correspondent. It is further reduced to correctness in the single-thread case due to the independence among all threads. During the debugging phase, this correctness relies on the equivalence between the multi-threaded version of the program (running with a single thread) and the original sequential program. This can be checked by a comparison based on a bi-simulation tool. During the production-run phase, this correctness relies on thread independence — a characteristic that can be verified by DRC with negligible overhead when the transformed code is correct.
4. **Transformation Scalability** is guaranteed by eliminating performance bottlenecks particular to multi-threaded C/C++ programs: memory allocation and implicitly shared variables. The memory allocation bottleneck is solved by TPEmalloc. Detection of write access to implicitly shared variables is targeted by DRC, which can find offending code even when running multi-threaded code on a single core.

3.1 Code Transformation

There are two steps of code transformations: the first step transforms the source code to replace several independent copies of the sequential process with an equivalent single process with several threads; the second step enables threads to use many-core machine in a memory-efficient way.

3.1.1 **TTS: Transformation for Thread Safety**

This step of transformation includes two parts: global variable collection and global variable privatization. The privatization is implemented via the ANSI C/C++ keyword `thread` (since C99). Correspondingly, all declarations qualified for this thread local storage (TLS) keyword should be collected.

A rigorous way to collect the information for all global variables is to patch some code in the C++ parser to recognize them. Taking the GNU (GCC version 4.2.2 [51]) C++ parser as an example, we change the source file `parser.c` to patch some output statements there. As described by the box “Patched Parser” in Figure 1, the patched parser does not change the compilation much. It merely collects all global declarations and corresponding extern declarations; and static declarations as a side effect.

Recent GCC compiler versions (since version 4.5) support plug-ins, that leads to a portable solution to collect qualified variables. One future solution will be a GCC plug-in utility on the application level. This plug-in program is triggered after several phases of compilation to filter out the qualified variables, as indicated by Figure 1. This solution is similar in spirit to the static analysis tool, Dehydra [52], and the LLVM background for GCC, DragonEgg [53]. DragonEgg provides alternatives for GCC’s optimizers and code generators.
<table>
<thead>
<tr>
<th>TTS.1</th>
<th><code>int global = 0;</code></th>
<th><code>-thread int global = 0;</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>TTS.2</td>
<td><code>static nonPOD field;</code></td>
<td><code>static _thread nonPOD *newfield;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>if (!newfield) //Patch as indicated</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>newfield = new nonPOD; //by the compiler</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>#define field (*newfield)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>#define CLASS::field (*CLASS::newfield)</code></td>
</tr>
<tr>
<td>TTS.3</td>
<td><code>static int var1 = var2;</code></td>
<td><code>static _thread int *var1_NEW_PTR_ = 0;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>if (!var1_NEW_PTR_)</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>{ var1_NEW_PTR_ = new int;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>*var1_NEW_PTR_ = var2; }</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>int &amp;var1 = *var1_NEW_PTR_;</code></td>
</tr>
<tr>
<td>TMR.1</td>
<td></td>
<td><code>//dynamically extended by the main thread</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>//and replicated by worker threads</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>class volume</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>{ //large relatively read-only field</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>RD_t RD;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>//small transitory member field</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>RDWR_t RDWR; }</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>_thread vector&lt;volume*&gt; store;</code></td>
</tr>
</tbody>
</table>

Table 1: Representative Code Transformation for TTS and TMR

To each qualified declaration, we add the ANSI C/C++ keyword `_thread`, as described by TTS.1 in Table 1. This step corresponds to the box “Thread Privatization” in Figure 1. After this step, at the binary level, the data segment is almost empty with merely some `const` values left. As a result, each thread behaves almost the same as the original process, since any variable that could have been shared has been declared thread-local. Naturally, the transformed code is thread-safe.

The implementation for the TTS thread privatization is based on an open source C++ parser, Elsa [54]. The thread privatization for variables takes advantage of thread-local pointers, that serves as a remedy for TLS to support variables that are not plain old data structure (not `POD`) and to implement dynamic initialization for TLS variables. Two examples TTS.2 and TTS.3 in Table 1 demonstrate the implementation.

3.1.2 TMR: Transformation for Memory Footprint Reduction

In transformation TMR, we determine relatively read-only field members, segregate and replicate other field members; and remove some `_thread` keyword so that threads can share part of data. The left part of TMR.1 in Table 1 is an example to illustrate the motivation of TMR. Speculatively, the class “volume” is a `shareable class` that contains some relatively read-only fields, whose total size is large. Moreover, this class may also contain some transitory fields, whose total size is conjecturally small. The TMR’s goal is to make threads share relatively read-only fields meanwhile replicate transitory fields. However, it is not clear which field is relatively read-only and which field is transitory.

The first step of TMR is to figure out for each shareable class relatively read-only member fields.
Valgrind’s Helgrind [55] module is an optional method for this purpose. To expose transitory data, one can deliberately start two threads to pass through the same task individually yet one-after-one. If any transitory data is changed, it must be changed at least once by both threads. Observing two threads change the same memory unit without locking, Helgrind must complain and output the error message to indicate the instruction and the address of the variable.

However, the optional method does not fit for large applications because Helgrind will make data initialization too slow to finish. Instead, the write access reflection based on DRC is recommended as a feasible method. This method includes two parts:

1. Framework code to overload the “new” and the “delete” methods for sharable classes and their container allocators

2. Auxiliary program (MWPT) to direct the execution of the given program

The framework code facilitates modifying all sharable classes to allocate their instances in a pre-allocated region in the heap. The pre-allocated region will be write-protected during the computation for tasks. MWPT uses the ptrace system call to direct the execution of the given program similarly to how GDB [56] interacts with an inferior. In addition, MWPT catches segfault signals then recognizes transitory member fields.

The second step of TMR enables threads to share instances whose transitory member fields have been moved to the thread-private region. The implementation is described by the right part of TMR.1 in Table 1, whose objective is illustrated by Figure 2.

In this figure, two threads share three instances for the sharable class “volume” in heap. The transitory member field set RW-Field has been removed from the class “volume”. A new field is added as instance ID and declared by the line 3 of TMR.1. In Figure 2, the ID for each shared instance is 0, 1 and 2 respectively. Each thread uses a TLS pointer to an array of RW-Field, that is indexed by instance ID. The RW-field array is declared by line 1 while the RW-field reference is redefined by the line 5 macro correspondingly. As we can see from Figure 2, when the worker thread 1 accesses the RW-Field set of instance 0, it follows the TLS pointer and assigns the RW-
Field to 1. Similarly, the worker thread 2 follows the TLS pointer and assigns the RW-Field to 8. Therefore, two threads access different memory units for the RW-Field in the same instance.

3.2 Transformation Correctness

The STOCT correctness is embodied by three requirements: (i) each thread should process each task just as the original program does for the corresponding task; (ii) the computation of each thread should never “collide” with the computation of any other thread; and (iii) unintended “collision” must be coordinated to satisfy task atomicity.

3.2.1 Bi-Simulator: Debug for Single-Threaded Correctness

TMR may introduce design fault that produces errors during thread initialization. We pin down this kind of error by comparing the transformed program with the original version to locate the injected bug that differentiates two version of program. The comparison verifies the original program and the multi-threaded version against whether they behave identically. This is exactly the proof via the bi-simulation in which the original program is regarded as the “specification” and the multi-threaded version is regarded as the “implementation”. Furthermore, the problematic instruction that leads to a difference is the clue for the root cause of the bug.

Bi-Simulator adopts a temporal approach and integrates DMTCP [57, 58] to facilitate the debugging by comparison. It employs a temporary reversibility similar to the general “replay” technique in some other debugging tools [59, 60, 61, 62]. However, Bi-Simulator simplifies this technique to a single-thread case where determinism is guaranteed embarrassingly. To explain the idea of the debugging by comparison, we introduce some symbols:

- $M_o(0)$: the heap data of the original program after initialization
- $M_o(i)$: the heap data of the original program after $i$ steps of execution
- $M_p(0)$: the heap data of the work thread after initialization
- $M_p(i)$: the heap data of the work thread after $i$ steps of execution
- $I_o(i)$: the $i^{th}$ instruction of the original program
- $I_p(i)$: the $i^{th}$ instruction of the work thread

Moreover, we have symbol pairs

- $(M_o(i - 1), I_o(i))$ and $(M_p(i - 1), I_p(i))$: the computation of the $i^{th}$ step for two versions of programs respectively
- $(n_o(i), v_o(i))$ and $(n_p(i), v_p(i))$: the representative data item of $M_o(i)$ and $M_p(i)$ respectively

The computation satisfies that

- $(M_o(i - 1), I_o(i)) \models M_o(i)$
- $(M_p(i - 1), I_p(i)) \models M_p(i)$
In addition, the task-oriented feature requires

- \( \forall i, M_o(i - 1) \equiv M_p(i - 1) \rightarrow I_o(i) \equiv I_p(i) \) and
- \( M_o(i) \equiv M_p(i) \).

Whenever an error occurs, i.e., \( \exists d, M_o(d) \neq M_p(d) \), we must have \( M_o(0) \neq M_p(0) \). Furthermore, the root cause is: tuples \( (n_o(0), v_o(0)) \in M_o(0) \) and \( (n_p(0), v_p(0)) \in M_p(0) \) such that \( n_o(0) \equiv n_p(0) \) but \( v_o(0) \neq v_p(0) \). To debug this root cause, Bi-Simulator is used as follows:

1. **Check whether** \( I_o(1) ... I_o(d) \equiv I_p(1) ... I_p(d) \). This is a typical binary search algorithm. Started from \( M_o(0) \) and \( M_p(0) \), execute \( d/2 \) steps. If they stop at the same \( I_o(d/2) \) and \( I_p(d/2) \), then recursively check execution steps of \([d/2 + 1, d]\). Otherwise, recursively check execution steps of \([1, d/2]\). We’d like to find the minimal \( d' \) such that \( I_o(d') \) and \( I_p(d') \) are the same conditional branch instruction but \( I_o(d' + 1) \neq I_p(d' + 1) \). Because the design fault introduced by TMR is generally serious and dominant, this check is already sufficient when the branch difference indicates the root cause of the error.

2. **Search the minimal** \( d' \) **such that** \( (n_o(d'), v_o(d')) \in M_o(d') \), \( (n_p(d'), v_p(d')) \in M_p(d') \), \( n_o(d') \equiv n_p(d') \) and \( v_o(d') \neq v_p(d') \). This is necessary if and only if \( I_o(d' + 1) \neq I_p(d' + 1) \) fails to reveal the root cause of the error, a situation that demands a further comparison of \( I_o(1) ... I_o(d') \) and \( I_p(1) ... I_p(d') \) given they are identical. The search algorithm is to step through each pair of \( I_o(i) \) and \( I_p(i) \), and to compare left-hand values in the case of assignment declarations.

### 3.2.2 DRC: Data Race Coordinator for Runtime Correctness

The source-code transformation has the potential to create data races in the resulting code when the cache coherence principle of Section is violated. To handle this kind of data race, STOCT addresses the run-time correctness, as described in Section 1. This correctness relies on atomicity of tasks as part of STOCT’s task-oriented nature.

The above type of correctness is strictly stronger than the more pragmatic *application-level correctness* introduced in [63]: if the developer accepts the results as correct, then the program is correct. Moreover, the run-time component of STOCT serializes tasks if any violation of the STOCT prerequisite (Section ) occurs and will retry each task that might be injured.

The memory write-protection technique allows DRC to serve as a run-time verifier to decide whether any shared data has ever been changed or not. Furthermore, DRC guarantees a correct production run due to its *recovery strategy*. Figure 3 briefly illustrates the protocol between DRC and the multi-threaded version for the given program.

1. The multi-threaded program (the “inferior” process in the terminology of ptrace) sets up signal handlers and spawns MWPT (the “superior” process in the terminology of ptrace).
2. The superior attaches and notifies the inferior to remove the “write” permission from the shared memory region.
3. The inferior processes tasks as usual until a task writes to the shared data region, that will generate a segfault.
4. The superior intercepts and relays the segfault to the inferior to re-enable the “write” permission for the shared memory region, retry the instruction which causes the segfault, and return to step 2.

5. After all tasks have been processed, the inferior will actively re-enable the “write” permission for the shared memory region and tell MWPT to terminate, which then forces MWPT to detach.

The superior controls the inferior in a per-thread fashion. The recovery strategy is implemented by changing step 4 mentioned above to suspend each thread that is trying to update the shared memory region. In that task all remaining threads finish their current task and arrive at a quiescent state. Then, all quiescent threads wait upon the suspended threads. The superior first picks a suspended thread to resume and finish its current task. All suspended threads then redo their current tasks in sequence.

3.3 Transformation Scalability

The STOCT scalability does not consider any improvement obtained by migrating from distributed memory to shared memory: communication cost reduction, system resource sharing or parallel algorithm refinement. It focuses on the elimination of shared memory performance bottlenecks.

3.3.1 TPMALLOC: Custom Scalable Malloc Library

Most memory allocator implementations maintain memory chunks using a “boundary tag” method as originally described by Knuth [64]. Consecutive memory allocations and deallocations lead to random accesses to memory space by tagging the head and the tail for every chunk. Although memory allocation and deallocation consist of 10 to 20 instructions only, the random pattern for memory accesses brings more cache misses to the malloc library and the corresponding slowdown. The negative influence of these cache misses will be mitigated by practical applications that
immediately access the first several bytes in the allocated memory chunk. However, these cache
misses increase the proportion of cost occupied by allocations/deallocations. Therefore, the cost
for memory allocator is not as negligible as it looks like.

The malloc standard requires the use of a shared memory data structure (arena) so that any
thread can free the memory allocated by any other thread. By Theorem 1.1 in Section 1, writes to
shared arenas experience more cache misses when the number of threads is large. These additional
cache misses increase allocation’s cost percentage further, that is the reason for the performance
degradation from some allocator implementations. A more surprising observation is that purely
simultaneous memory allocations/deallocations have a larger wall-clock time than a corresponding
sequential execution does, when the number of threads goes above 8.

**Example.** Consider the glibc standard malloc library ptmalloc2, this slow-down is reproducible
through a toy program in which multiple threads work cooperatively on a fixed pool of tasks. The
task for the toy program is to allocate 4,000 chunks of size 4 KB and to then free them. As the
number of threads goes from 8 to 16, the wall-clock time increases, even though the load per thread
is halved. The reason for the slowdown could be studied via “strace”, that exposes a huge number
of “futex system calls”, ordinarily generated by locks&unlocks. The address statistics for those
system calls demonstrated that most of them came from the malloc library. In addition, more
cache misses are observed with 16 threads.

Some other malloc implementations such as tcmalloc, ptmalloc3 and hoard improve the per-
formance for the toy program. However, they fail to solve the performance bottleneck for some
extreme applications [5].

One recent work [65] exploits the internal parallelism in the malloc library to speed up alloca-
tions/deallocations. This work introduces a memory management thread (MMT) with an efficient
design, that improves the performance for applications where the security check overheads are in-
tensive. The experimental results demonstrates a huge reduction for the security-check overhead
and achieves an overall speedup ratio of 1.19 for two allocators.

The preferred solution is a non-standard extension to a standard allocator implementation,
TPMalloc, that associates a thread-private malloc arena to each thread. This makes a non-portable
assumption that if a thread allocates memory, then the same thread will free it. We call the memory
region associated with this pair of malloc and free as *transient object*.

A thread-local global variable is also provided, so that the modified behavior can be turned on
or off on a per-thread basis. By default, threads use thread-private malloc arenas. In portions of
the code where programmers know that a thread executing that code will need a shared central
heap region for non-transient objects, they switch off/on the thread-local global variable.

As Figure 2 demonstrates, this allows the multi-threaded program to keep both transient objects
and transitory data in a thread-private heap region. Therefore, the original lock associated with
each arena in the standard implementation, is no longer used by the custom allocator.

**Example.** Generally, large C++ programs are full of containers and string declarations [66], which
implicitly leads to intensive use of memory allocation and deallocation. If this intensive memory
allocation resides in a “hot” function, then the time for allocation/deallocation will dominate.
Under such circumstances, the centralized heap of a traditional memory allocator is overkill, because
those implicit allocations and deallocations are thread-private. In contrast, TPMalloc provides a
memory allocation library that scales well for many threads.

14
### Table 2: Typical C++ Source for Shared Updates

<table>
<thead>
<tr>
<th>Code Block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cout.precision(*)</code></td>
<td>Obvious shared update</td>
</tr>
<tr>
<td><code>str = &quot;&quot;</code></td>
<td>All empty strings refer to the same static instance with a reference count. This assignment changes the reference count.</td>
</tr>
<tr>
<td><code>std::ostringstream os;</code></td>
<td>The default constructor for std::ostringstream takes a static instance of the locale class, that changes the reference count for this instance.</td>
</tr>
</tbody>
</table>

#### 3.3.2 Reduction for Extraordinary Cache Misses

Theorem 1.1 with the formula from its proof reveals that synchronization on updates to shared variables while maintaining cache coherence generates an extraordinary number of cache misses. If updates to shared variables is the primary source of cache misses, then the analytical formula for the number of extraordinary cache misses can be relied upon to correctly predict the number of updates to shared variables. The formula will be used in two ways.

1. The formula will be used to confirm that most of the remaining cache misses are due only to updates to shared variables. This is done by considering different cases from the measured data, and showing that all of the legal $w$ values predict approximately the same number of shared variable updates.

2. Given the predicted number of shared variable updates, this is used to determine if the number of shared variable updates is excessive. When can we stop looking for shared variables that are frequently updated? Answer: When the number of shared variable updates is sufficiently small.

This formula contributes to an indirect method for DRC to pin down extraordinary cache misses due to shared memory updates. The DRC-based method attributes those cache misses to corresponding instructions by protecting more regions to recognize write accesses, as seen in Figure 2. DRC regards shared updates as the root cause for unscaleable expressions, whereas a scalable expression is defined as: given a workload that incurs many evaluations of the expression in question, a parallel evaluation with $k$ threads on $k$ cores spends $1/k$ of the time that one thread with the same workload.

**Example.** If threads have many unscaleable expressions, performance profilers such as pfmon [67] will report excessive cache misses. DRC is effective in exposing the root cause for those cache misses. Table 3.3.2 presents three examples of frequently used C++ expressions. The reference counting mechanism in GNU C++ renders these expressions unscaleable.

Whether there is any shared update and which variables might be shared all depend on the particular multi-threaded application, including the particular libraries invoked. It is infeasible for human being to go through a large application and to recognize all potential shared variable updates. Instead, DRC can be employed to locate unscaleable instructions whenever updates for shared variables occur implicitly and intensively.
4 Applicable Scenario for STOCT

Generally, the system’s C library is thread-safe. However, the C++ standard does not address matters of multi-threaded applications. For multi-threaded applications which use standard C++ containers, the proposed STOCT addresses to replicate those C++ data structures for each thread if they are not relatively read-only. In the case threads really need to share a C++ container instance from the standard template library (STL), STOCT can also work around a sharing mechanism by recognizing update instructions to this instance. As an example, one may work around a shared C++ STL hash table which is updated by threads occasionally. Three more example applications are also considered for which the STOCT methodology is applied.

4.1 Parallel LU Factorization and ILU(k) Preconditioning

LU factorization is an important algorithm in matrix computation. It is widely used to solve large sparse systems of linear equations of the form $Ax = b$, where $A \in \mathbb{R}^{N \times N}$ is the sparse matrix, $b \in \mathbb{R}^N$ is the right hand-side vector, and $x \in \mathbb{R}^N$ is the vector of unknowns of which a solution is sought. Generally, LU factorization-based methods [68] stores large sparse matrices in the compressed sparse row (CSR) format or in the compressed sparse column (CSC) format. In addition, address space needs to be reallocated dynamically for fill-ins no matter a direct method or an iterative method is adopted. STOCT benefits the parallelization [6] for this algorithm in three ways:

1. TPMalloc (Section 3.3.1) eliminates the performance bottleneck for the allocator;
2. TTS (Section 3.1.1) recognizes all global declarations and prevents potential data races;
3. DRC (Section 3.1.2) guarantees that the data segment usage does not produce a data race.

4.2 Geant4 Multi-threading

Geant4 [69, 70] was developed over about 15 years by physicists around the world, using the Booch software engineering methodology. It is widely used in high energy physics [71], radiation-based medical research [72], cosmic ray simulation [73], and space and radiation simulation [74]. Geant4 consists of 750,000 lines of C++ code spread over 6,000 files with deep knowledge of the physics of particle tracks.

As a Monte Carlo simulation toolkit, Geant4 profits from improved throughput via the parallelism derived from the independence among modelled events and their computation. In the era of computer clusters, researchers have implemented ParGeant4 [75] for a master-worker style event-level parallel computing on distributed-memory multiprocessors.

Many-core computing has gained an increasing presence in the landscape. Targeting this new platform, STOCT transforms Geant4 into a multi-threaded version following the same event-level parallelism as the prior distributed memory parallelization has done. The multi-threaded Geant4 achieves linear speedup in a range from one thread to 24 on a 24-core computer while per thread memory footprint reduced to a small constant and is less than 10 percent of total application data.
4.3 AliRoot Multi-threading

AliRoot [7] is also a million line C++ framework based on ROOT [76]. AliRoot uses an object-oriented framework for large scale data analysis. To avoid an excessive memory footprint, AliRoot stores data to and loads data from files. Even so, AliRoot still requires about 3 GB memory for data processing. On an eight-core machine, the total memory consumption would be 24 GB for eight AliRoot processes. Since the number of cores per machine is growing with Moore’s Law, it is necessary to replace AliRoot processes with equivalent threads, which share both memory and secondary storage.

5 Schedule of Milestones

The following gives anticipated completion dates for a series of milestones in the proposed thesis work.


References

[23] Christopher Grant Jones and Rose Liu and Leo Meyerovich and Krste Asanović and Rastislav Bodík: Parallelizing the Web Browser. In: USENIX HotPar’09 (2009)


