Composing Locks by Decomposing Deadlocks

Hari K. Pyla
Virginia Tech
harip@vt.edu

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; D.3.4 [*Programming Languages*]: Processors—Runtime environments; D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures

General Terms Algorithms, Design, Languages, Measurement, Performance and Reliability

Keywords Concurrent Programming, Runtime Systems, Program analysis, Deadlock Detection and Recovery, Speculative Parallelism and Coarse-grain Speculation

1. Introduction

The evolution of processor architectures from multi-core to many-core requires programmers to use concurrency to achieve performance. Unfortunately, shared memory parallel programs are difficult to implement correctly, and so is detecting concurrency bugs (e.g., data races, deadlocks, order violations, atomicity violations). In practice, the most common concurrency bugs are a) data races that arise due to unguarded or improperly guarded memory updates and b) deadlocks that arise due to circular dependencies among locks. While data races can be ameliorated by appropriate synchronization (a challenging problem in itself), deadlocks require fairly complex deadlock avoidance techniques, which may fail when the order of lock acquisitions is not known a priori. Furthermore, due to the potential for deadlocks, programmers cannot arbitrarily compose lock based codes without knowing the internal locking structure. Hence, *composability* is limited by deadlocks. *The goal of this research is to achieve composability of lock based codes*.

We present Sammati [1] (*agreement* in Sanskrit), a software system that is capable of *transparently* and *deterministically* detecting and recovering from deadlocks in multi-threaded applications, without requiring any modifications to application source code or recompiling/relinking phases. Sammati is implemented as a pre-loadable library that overloads the standard POSIX threads (pthreads) interface and supports applications written using weakly typed languages such as C and C++. It guarantees the acquisition of mutual exclusion locks a deadlock free operation. Sammati supports arbitrary application level threading models, including those that use locks for concurrency control where serial lock elision does not result in a program with the same semantics.

Sammati associates the memory accesses with locks and privatizes memory updates within a critical section. The updates within a critical section are made visible outside the critical section on the release of the parent lock(s), viz. the containment property. On the acquisition of every lock, Sammati checks for deadlocks. If a deadlock is detected, the deadlock elimination algorithm breaks the cycle by selecting a victim, rolls it back to the acquisition of the offending lock, and discards its memory updates. Since our containment mechanism ensures that memory updates from a critical section are not visible outside the critical section until a successful release, we simply restart the critical section to recover from the deadlock.

Although the core idea behind Sammati is quite simple, there are several challenges in the details of this work. First, we need to provide a transparent mechanism for detecting memory updates within a critical section and privatizing the updates. Second, in the context of nested locks we need to define a set of visibility rules that preserve existing lock semantics, while still permitting containment based deadlock elimination and recovery. Finally, we need a deadlock detection and recovery mechanism that is capable of deterministically eliminating deadlocks without either (a) deadlocking itself or (b) requiring an outside agent. Additionally, Sammati can detect and report *write-write* races that occur between (a) guarded and concurrent unguarded updates to a shared value and (b) improperly guarded updates, where a single data value is guarded by two or more different locks. In this research we propose and implement techniques that address the above design objectives.

We evaluated its performance of Sammati using SPLASH, Phoenix and synthetic benchmark suites on a 16 core shared memory machine (NUMA) running Linux with 64GB of

SPLASH'11 Companion, October 22–27, 2011, Portland, Oregon, USA.
ACM 978-1-4503-0940-0/11/10.

RAM. We measured the number of locks acquired and lock-acquisition-rate (total locks acquired/total runtime) for all applications used in this study. While Sammati's runtime is impacted by lock acquisition rate, it shows speedup comparable to the native pthreads case even for applications that have large ($\approx 89,500$ locks/sec) lock acquisition rates. This is in contrast to transactional memory systems, which have significant impact on speedup, largely due to privatization at the instruction level and the need to guard every read from read/write conflicts. Additionally, the space overhead of our approach is O(W), where W is the write set (in pages) within a lock context. Finally, we also evaluated Sammati by running programs that were deadlock prone. We find that the native pthreads programs deadlock while Sammati deterministically detects and avoids the deadlocks, transparently recovers from them and successfully executes the program to completion. Our results indicate that for most applications the speedup of Sammati is comparable to that of native Pthreads with modest memory overhead.

Contributions and Impact

There are several aspects of this work that will significantly impact the usability of lock based programming for concurrency on multi-core architectures.

- **Handling Deadlocks:** Existing systems rely on program analysis, modifications to source code and/or operating system and, prediction techniques to identify the occurrence of deadlocks. In contrast, Sammati deterministically detects and recovers from deadlocks at runtime without requiring access or modifications to source code in applications with arbitrary threading models. Our proposed approach readily enables its use with existing applications. Additionally, Sammati's language transparency, enables its use with a wide variety of programming languages including unmanaged languages such as a C, C++ and Fortran.
- **Programmer Productivity:** Due to the non-deterministic nature of thread execution, it may not be feasible in practice to verify and test all possible interleavings of threads and their lock acquisitions in order to determine if a program is deadlock free. Programmers can write code to the best of their ability and rely on the runtime system (Sammati) to handle deadlock detection and recovery.
- **Mutual Exclusion Locks:** Most programmers are already familiar with lock-based programming as opposed to using transactional memory systems and addressing issues in lock based programming can benefit the large base of lock-based software artifacts in use today. Previously, lock based codes were not composable due to the potential of deadlocks –one of the primary motivations for transactional memory. Since Sammati provides a robust mechanism to address this problem, we believe that this work will enable a new generation of composable lock-based codes.

2. Current and Future Directions

Several novel techniques proposed in the research opens way to solve important challenges faced in concurrent programming.

2.1 Program Analysis and Shadowing Memory

We are currently working on improving the performance of Sammati through compile time analysis and instrumentation. Sammati's overhead primarily stems from the protection and privatization of the virtual address space. We believe that we can reduce this runtime overhead by employing program analysis to accurately determine the write-set (i.e., data modified) within a lock even in the presence of nested and conditional lock acquisition and release sequences. There are several challenges in the details of this work. First we need a mechanism to identify locks and their scope in the program. Second, we need to accurately determine the write-set (i.e., data modified) within a lock. In situations where program analysis cannot determine control flow, the Sammati runtime can act as the fail-safe to provide deterministic deadlock detection and recovery. Third, we need to isolate the memory updates within locks to facilitate recovery on deadlock. We need a lightweight memory shadowing mechanism to accomplish isolation. Additionally, the ordering and integrity of the load and store instructions must be preserved in order to maintain program correctness. We plan on leveraging the LLVM compiler infrastructure to implement some of our proposed techniques. We will be evaluating our approach using SPLASH, Phoenix and PARSEC benchmarks.

Given the scope of this research, we expect the following key research deliverables from this work. First, a runtime infrastructure for transparent deadlock detection and recovery for POSIX threaded codes with any threading model, which enables composability of arbitrary lock-based codes. Second, reachability and flow analysis methods to minimize the performance impact of deadlock detection and recovery mechanisms in the common case of deadlock free operation. Third, novel methods to transparently eliminate priority inversion problems in threaded codes operating in real-time infrastructures. Fourth, mechanisms to support non-idempotent operations such as memory management and IO within critical sections that may be affected by deadlock recovery. Fifth, methods to guide deadlock victim selection based on a variety of performance and correctness metrics and techniques to ensure safe progress when threads abort while holding locks. To our knowledge, several of the proposed research deliverables present the only known solutions to the corresponding research problem and will complement existing research in the area of compilers and runtime systems.

2.2 Concurrency Bugs and Managed Languages

We are currently extending our work to detect other forms of concurrency bugs (e.g. data races) and provide concurrency bug detection and composability of lock based codes in managed languages such as Java.

2.3 Support for Speculative Execution

The impending multi/many core processor revolution requires that programmers leverage explicit concurrency to improve performance. Unfortunately, a large body of applications/algorithms are inherently hard to parallelize due to execution order constraints imposed by data and control dependencies or being sensitive to their input data and not scale perfectly, leaving several cores idle in the impending multi/many-core processor revolution. *The goal of this research is to enable such applications leverage multi/many-cores efficiently to improve their performance*. Our objective in this work is extend the lock/unlock semantics of Sammati to begin/commit/abort semantics and to provide programmers with a tool for exploiting parallelism in such applications.

Technical Approach

This work equips programmers with a powerful tool for exploiting parallelism by means of *coarse-grain speculation*. Our programming model can express computation at any granularity, so that any application unit can be executed speculatively without burdening the programmer from the subtleties of concurrency programming such as using the low level threading primitives to create speculative control flows, manage rollbacks, and recover in the event of misspeculations.

We present a simple speculative programming framework, Anumita (*guess* in Sanskrit) [2], in which coarse-grain speculative code blocks execute concurrently, but the results from only a single speculation modify the program state. Anumita is implemented as a shared library that exposes APIs for common type-unsafe languages including C, C++ and Fortran. Its runtime system transparently (a) creates, instantiates, and destroys speculative control flows, (b) performs name-space isolation, (c) tracks data accesses for each speculation, (d) commits the memory updates of successful speculations, and (e) recovers from memory side-effects of any mis-predictions.

Anumita associates each speculation flow's (e.g., an instance of a code block or a function) memory accesses in a speculation composition (loosely, a collection of possible code blocks that execute concurrently) and localizes them, isolating speculation flows through privatization of address space. Ultimately, a single speculation flow within a composition is allowed to modify the program state. We present well-defined semantics that ensures program correctness for propagating the memory updates. Anumita supports a wide range of applications by providing expressive evaluation criteria for speculative execution that go beyond *time to solution* to include arbitrary *quality of solution* criteria.

Using Anumita requires minimal modifications (8-10 lines on average) to application source code. Additionally, the speculation-aware runtime manages memory and collects garbage from failed speculations. In the context of high-performance computing, with the prevalent OpenMP threading model, Anumita naturally extends speculation to an OpenMP context through a pragma. To our knowledge, Anumita is the first system to provide support for exploiting coarse-grain speculative parallelism in OpenMP based applications.

Our preliminary results [2] using real applications such as graph coloring problem, partial differential equation (PDE) solvers and combinatorial problems including sorting indicate that Anumita is capable of significantly improving the performance of hard-to-parallelize and input sensitive applications by leveraging speculative parallelism. For instance, in the PDE solver the speedup ranged from 0.84 to 36.19, for the graph coloring problem it ranged from 0.95 to 7.33, and for the sort benchmark it ranged from 0.84 to 62.95. Using Anumita it is possible to obtain the best solution among multiple heuristics. We found that in some cases where heuristics failed to arrive at a solution, the use of speculation guaranteed not only a solution but also the one that is nearly as fast as the fastest alternative. Anumita's preliminary results indicate that it is possible to exploit coarse-grain speculative parallelism without sacrificing performance, portability and usability.

3. Summary

In this dissertation we provide novel techniques to solve several challenges faced in concurrent programming. The contributions of the research will extend well beyond our core technical contributions. By providing usable and efficient deadlock detection and recovery for threaded codes, we will provide a critical tool to programmers designing and implementing (and debugging!) complex applications for emerging many-core platforms. In addition, our solutions to several hard problems in concurrent programming will each serve to broaden the set of codes, and even application domains that will benefit from the rise of manycore platforms. More broadly yet, this dissertation will impact the future of concurrent programming and assist in improving the productivity of application developers. We believe that our research efforts will help adapt and sustain the increasing core counts of multi/many-core systems.

References

[1] H. Pyla and S. Varadarajan. Avoiding Deadlock Avoidance. In PACT '10, pages 75–86, New York, NY, USA, 2010.

[2] H. Pyla, C. Ribbens, and S. Varadarajan. Programmable Coarse-Grain Speculative Execution. In OOPSLA '11, New York, NY, USA, 2011.