Transparent Runtime Deadlock Elimination

Hari K. Pyla
Virginia Tech
Blacksburg, Virginia, United States
harip@vt.edu

Srinidhi Varadarajan*
Virginia Tech
Blacksburg, Virginia, United States
srinidhi@vt.edu

ABSTRACT

Thread based concurrent programming is hard due to the potential of concurrency bugs (e.g., data races, atomicity violations, deadlocks, and order violations). While data races and atomicity violations can be ameliorated with appropriate synchronization (a non-trivial problem in itself !), deadlocks require fairly complex avoidance techniques which may fail when the order of lock acquisition is not known apriori [2]. *The goal of this research is to present an efficient and practical system that transparently detects and eliminates deadlocks in real-world multi-threaded applications.*

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.4 [**Programming Languages**]: Processors—*Run-time environments*

General Terms

Algorithms, Design, Languages, Performance and Reliability

Keywords

Deadlock Detection and Recovery, Concurrent Programming, Program Analysis, and Runtime Systems

1. INTRODUCTION

In practice, identifying a concurrency bug does not necessarily mean that it can be easily fixed. To properly fix the concurrency bug, the programmer must identify the root cause of the problem rather than simply observing how the bug manifests itself in the program's execution. Additionally, such bugs are often hard to reproduce. Properly fixing a concurrency bug may require a major software redesign. Recent studies have shown that the patches developed to fix a bug are themselves error-prone (70% of the time in their first release) and they introduce new bugs [1]. On average, a concurrency bug fix takes about 3 fixes (patches) before it is actually fixed [3, 1]. Unless we find mechanisms to enable a large number of programmers, representing a wide array of applications, to use these parallel shared memory platforms effectively, the potential of many-core will go unrealized.

In this research we present **Serenity**, a system that transparently and deterministically eliminates deadlocks at runtime in applications written in type-unsafe languages such as C, C++. Serenity operates by (a) associating memory updates with one or more locks guarding the updates and (b) containing (privatizing) the updates until all locks protecting the updates have been released. All memory updates within a critical section protected by one or more locks are performed atomically at the release of all surrounding locks. In such a system, deadlock detection can be performed at the acquisition of each lock and recovery merely involves selecting a victim lock and discarding all privatized memory updates performed subsequent to the acquisition of the victim.

Serenity transforms the program source to LLVM's IR and provides (a) efficient programming analysis techniques that infer the scope of a lock and (b) selective compile time instrumentation of the identified scope. Serenity's runtime system provides an efficient runtime shadowing technique to implement containment, thereby transparently and automatically eliminating deadlocks.

A comprehensive performance analysis using several applications shows that the scalability of Serenity is comparable to native thread execution with modest performance overhead.

The key contribution of this work is a practical system that can eliminate deadlocks in real-world applications. By providing usable and efficient deadlock detection and recovery for threaded code, Serenity provides a critical tool to programmers designing, implementing and debugging complex applications for emerging many-core platforms.

*Faculty Advisor

PACT'12, September 19–23, 2012, Minneapolis, Minnesota, USA.

2. REFERENCES

[1] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, Automated Atomicity-Violation Fixing, In PLDI '11, pages 389–400.

[2] H. K. Pyla and S. Varadarajan, Avoiding Deadlock Avoidance, In PACT '10, pages 75–86.

[3] H. Volos, A. J. Tack, M. M. Swift, and S. Lu, Applying transactional memory to concurrency bugs, In ASPLOS '12, pages 211–222.