Science of Computer Programming 80 (2014) 223–263

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Static safety guarantees for a low-level multithreaded language with regions

Prodromos Gerakios [a], Nikolaos Papaspyrou [a,*], Konstantinos Sagonas [a,b]

[a] *School of Electrical and Computer Engineering, National Technical University of Athens, Greece*
[b] *Department of Information Technology, Uppsala University, Sweden*

HIGHLIGHTS

- Low-level language with hierarchical regions and reader/writer locks.
- Formal type and effect system with effect inference.
- Formalism and type safety proof: memory safety and race freedom.
- Design and integration into Cyclone.
- Performance evaluation against C using challenging benchmarks.

ARTICLE INFO

Article history:
Received 22 June 2011
Received in revised form 2 May 2013
Accepted 14 June 2013
Available online 23 July 2013

Keywords:
Safe multithreading
Type and effect systems
Region-based memory management
Cyclone

ABSTRACT

We present the design of a formal low-level multithreaded language with advanced region-based memory management and thread synchronization primitives, where well-typed programs are memory safe and race free. In our language, regions and locks are combined in a single hierarchy and are subject to uniform ownership constraints imposed by this hierarchical structure: deallocating a region causes its sub-regions to be deallocated. Similarly, when a region is read/write-protected, then its sub-regions inherit the same access rights. We discuss aspects of the integration and implementation of the formal language within Cyclone and evaluate the performance of code produced by the modified Cyclone compiler against highly optimized C programs using pthreads. Our results show that the performance overhead for guaranteed race freedom and memory safety is in most cases acceptable.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

With the emergence of commodity multicore architectures, exploiting the performance benefits of multithreaded execution has become increasingly important to the extent that doing so is arguably a necessity these days. Programming languages that retain the transparency and control of memory, such as C, seem best-suited to exploit the benefits of multicore machines, except for the fact that programs written in these languages often compromise memory safety by allowing invalid memory accesses, buffer overruns, space leaks, etc., and are susceptible to data races. Thus, a challenge for programming language research is to design and implement multithreaded low-level languages that provide static guarantees for memory safety and data race freedom and, at the same time, allow for a relatively smooth conversion of legacy C code to its safe multithreaded counterpart.

* Corresponding author.
E-mail addresses: pgerakios@softlab.ntua.gr (P. Gerakios), nickie@softlab.ntua.gr (N. Papaspyrou), kostis@cs.ntua.gr (K. Sagonas).

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2013.06.005

Towards this challenge, we present the design of a formal low-level concurrent language that employs advanced region-based management and hierarchical lock-based synchronization primitives. Similar to other approaches, our memory regions are organized in a hierarchical manner where each region is physically allocated within a single parent region and may contain multiple child regions. Our language allows deallocation of complete subtrees in the presence of region sharing between threads and deallocation is allowed to occur at any program point. Each region is associated with an implicit reader/writer lock. Thus, locks also follow the hierarchical structure of regions and in this setting each region is read/write protected by its own lock as well as the reader/writer locks of all its ancestors. As opposed to the majority of type systems and analyses that guarantee race freedom for lexically-scoped locking constructs [1–3], our language employs non-lexically scoped locking primitives, which are more suitable for languages at the C level of abstraction. Furthermore, it allows regions and locks to be safely aliased and to escape their lexical scope when passed to a new thread. These features are invaluable for expressing numerous idioms of multithreaded programming such as *sharing*, *region ownership* or *lock ownership transfers*, and *region migration*.

More importantly, our formal language is not just a theoretical design with some nice properties. As we will see, we have integrated our language constructs into Cyclone [4], a strongly-typed dialect of C which preserves explicit control and representation of memory without sacrificing memory safety. Long ago we opted for Cyclone because it is more than a memory-safe variant of C and its implementation is publicly available. Cyclone is also a low-level language that offers modern programming language features such as first-class polymorphism, exceptions, tuples, namespaces, (extensible) algebraic data types, and region-based memory management. We will discuss how these features interact with our language constructs and the additions that were required to Cyclone's implementation.

1.1. Contributions

This article combines ideas and material which we have presented in two workshop papers [5,6], but at the same time it significantly extends these works. In particular, our work on the formalization of hierarchical region systems [5] has laid the foundation for this article, albeit it did so with a type and effect system that is quite complicated and has several drawbacks: it requires *explicit* effect annotations, restricts aliasing, and allows temporary leaks of regions. Some of these drawbacks were lifted in the simpler and at the same time more refined type and effect system we subsequently developed [6], but its effect annotation burden was still quite high and made programming in Cyclone cumbersome. In addition, in this later system [6] region aliasing requires the programmer to create new capabilities, which entails a run-time overhead and makes programming less intuitive, and to use explicit count annotations as well as information about the "parent-of" relation, which limit polymorphism and result in code duplication.

In this article, we lift all these limitations. The type and effect system we will develop requires annotations *only* at thread creation points (i.e., at uses of the `spawn` operator) and all the remaining annotations are automatically inferred and checked by the analysis. Moreover, there are no annotations regarding region aliasing state (i.e., aliased and non-aliased regions). We also extend the formal language with permissions for *read-only* accesses to hierarchies. Such a feature is useful and increases concurrency when threads share regions without modifying them. Of course, a region can alternate between read-only and read/write or "no-access" states during its lifetime in a safe manner. The type system ensures this.

In short, the main features of the type system we present and the contributions of this article are as follows:

Hierarchical regions and reader/writer locks We develop a region-polymorphic lambda calculus, where regions are organized in a hierarchy and are protected with reader/writer locks. When a reader/writer lock of a region is acquired, then its subregions atomically inherit the same access rights. In addition, read/write-protected hierarchies can migrate or be shared with new threads.

Effect inference Functions need not be annotated with explicit effects and the system permits a higher degree of polymorphism as there are no explicit capabilities.

Formalisms and soundness We provide an operational semantics for the proposed language and a static semantics that guarantees absence of memory violations and freedom from data races. In addition, we state safety theorems and provide proofs for the soundness of the core language.

Design and implementation We discuss implementation issues related to static analysis, code generation and additions to the run-time system that were required in order to make the integration of the type system into Cyclone possible.

Performance evaluation We show the effectiveness of our approach by running benchmark programs.

1.2. Overview

The next section (Section 2) presents the design goals of our language and is followed by a brief section (Section 3) showing its main features by example. We then provide a description of the formal language, its operational semantics and static semantics (Section 4), followed by a section (Section 5) where the main theorems that guarantee the absence of memory violations and data races from well-typed programs are stated and proved. After briefly reviewing the Cyclone language (Section 6), we describe the integration of our language constructs into Cyclone (Section 7), followed by a presentation of implementation (Section 8) and performance (Section 9) aspects of this integration. The article ends by two sections discussing related work and containing some concluding remarks.

Download English Version:

https://daneshyari.com/en/article/6875371

Download Persian Version:

https://daneshyari.com/article/6875371

Daneshyari.com