CSDL Overview & Compiler Techniques to Improve Program Performance

by

Prasad A. Kulkarni
CSDL Lab at ITTC

• Focus on design, implementation and verification of computer systems

• People
  – Perry Alexander, Director
    • formal modeling of computer systems, security
  – Douglas Niehaus
    • real time and distributed systems, operating systems
  – Prasad Kulkarni
    • code optimization, parallelism, VM performance
  – Andy Gill
    • functional language design, extensions and implementation
Professional Background

• Experience
  – Assistant professor, EECS, University of Kansas (started in Fall 2007)
  – Intern, IBM T. J. Watson Research Lab (Fall 2006)

• Education
  – Ph.D. in computer science from Florida State University (Summer 2007)

• Research interests
  – compiler analysis & optimizations, profiling, architecture
  – to improve performance and security
  – in embedded, general-purpose, high-performance systems

• Teaching
  – compilers, operating systems, virtual machines
Recent Trends in Computer Systems

- Higher-level programming languages
- Increased Internet use & accessibility
- Safe & secure runtimes
- Growth in embedded systems
- Multicore architectures
Research Projects

• Improve performance and security of managed language programs on multicore machines
  – future profiling for virtual machines
    • improving Java start-up performance
  – parallelization model for virtual machines

• Performance of embedded system applications
  – understanding compiler optimization phase interactions
  – providing faster & effective phase ordering searches

• Compiler-based program parallelization
  – interactive generation of thread-safe programs
Future Profiling for Improved Virtual Machine Performance
Traditional Program Profiling

• Profiling
  – monitor and understand program behavior to improve program characteristics

• *Offline* profiling uses information from past runs

  + no runtime overhead
  – requires user to find representative input, perform profile run, encode/use information
  – reactive, fails if profile and current input do not match
Profiling in Virtual Machines

- **Online** profiling monitors the current run
  - no prior program run needed
  - can better adapt to changing input
  - need runtime system and causes overhead
  - still reactive?

- Java virtual machines use online profiling
  - during *selective* compilation
  - feedback-driven optimization
  - security checks
Profiling During Selective Compilation

• Current online profiling schemes are still *reactive*
  – employ very simple prediction models
  – future behavior is same as past behavior

• Leads to incorrect *speculations*
  – unnecessary compilation overhead at runtime
  – delays compilation of *actual* hot methods

• Optimizations wait until profiling results available
  – delays decisions based on profiling
  – degrades performance at program *start-up*
Improving Online Profiling – Hypothesis

• Profiling to understand *future* program behavior!

• For each online prediction task
  – construct program models
  – that use values of *key* variables
  – to *know* future program behavior
Parallelization Model for Virtual Machines
Inline Auxiliary Tasks

- Virtual machines perform several profiling tasks and security checks during program execution
  - profiling for improved performance
  - checks like taint propagation, on-access virus-scans
- Checks performed *inline* with the main program
  - introduces overhead

*Uniprocessor Program Flow*
Parallelizing Security Checks

Algorithm

1. program *slicing* to determine code statements necessary for each security check
2. minimize slice using other optimizations
3. factor out each security thread with its program slice into new *auxiliary* thread
4. Run auxiliary threads concurrently with main thread
Parallelizing Auxiliary Tasks

Uniprocessor Program Flow

Main Thread

Auxiliary Thread - 1

Auxiliary Thread - 2
Research Directions

- Techniques to find the smallest program slice
  - automatically determine the slicing criteria for different security constraints
- Slice representation in program binary
  - conserve binary size
- VM framework to concurrently execute auxiliary slices with main program thread
  - interpret new binary file
  - prevent auxiliary threads from changing global state
  - ensure correct program execution
Understanding Optimization Phase Interactions for Faster Phase Ordering Searches
Optimization Phase Ordering

• Changing order of phases affects code generated
  – large speed/size variations

• Current approach
  – optimization phases considered as black boxes
  – use heuristics to search *part* of phase order space

• Problems
  – optimal phase ordering not guaranteed
  – focus more on heuristic search techniques
  – no understanding of phase ordering issues
  – how to implement phases in future compilers?
Solution Approach

• Understand impact of registers on phase order space
  – explore techniques to reduce \textit{false} register dependences
  – copy propagation applied after every relevant phase reduces phase order search space by 27%, on average
  – locally remapping registers during optimizations improves performance by up to 14%

• Study partitioning of independent or cleanup phases
  – removing DAE from the search space reduces search space by 49% on average

• Generate rules for implementing phases at compiler build time
  – changing implementation later is difficult
Questions ?