Programming Systems for Heterogeneous Computing

Byeongcheol Lee
School of Information and Communications
Gwangju Institute of Science and Technology
Computer Ecosystem

Life

Data

Programs

Languages

Machines
Evolving Computer Ecosystem and Virtuous cycle

- Lots of features and fancy graphics in my new version of Matlab
Lots of features and fancy graphics in my new version of Matlab

But, it’s just too slow

Wait for a year and buy a faster hardware
Evolving Computer Ecosystem and Virtuous cycle

- Lots of features and fancy graphics in my new version of Matlab
- But, it’s just too slow
- Wait for a year and buy a faster hardware
- Run the program with lots of features on the top faster hardware
Virtuous cycle [ASPLOS 2013 Panel]

Doubling of transistors

Software Innovation
- Software Complexity
- Sequential Interface

Devices
- 2x more capable, efficient, cheaper, smaller, ...
- Hardware Complexity
- Sequential Interface
Breaks in virtuous cycle [ASPLOS 2013 Panel]

- Devices: 2x more capable, efficient, cheaper, smaller, ...
  - Doubling of transistors
  - End of Dennard Scaling

- Software: Innovation
  - Software Complexity
  - Sequential Interface

- Hardware: Complexity
  - Sequential Interface

Sequential Interface
Phase shift in the virtuous cycle

"Advancing Computer Systems without Technology Progress" [ASPLOS 2013 Panel]

General purpose DBMs
General purpose languages
General purpose CPUs
Dennard Scaling

Gone!

What do we want to do?
What do we have now?

System Capability (log)

CMOS

Fallow Period

Our Focus

New Technology

80s 90s 00s 10s 20s 30s 40s 50s

Fallow Period
Heterogeneous devices, languages, programs, and data

Heterogeneous computing devices
Heterogeneous devices, languages, programs, and data

Heterogeneous programming languages and models

Heterogeneous computing devices
Heterogeneous devices, languages, programs, and data

Heterogeneous programs and data

Heterogeneous programming languages and models

Heterogeneous computing devices
Heterogeneity in memory devices

Speed in time

On chip registers
SRAM
DRAM
SSD
Harddisk

Capacity in space

Cached, virtual memory
TLB & MMU
Demand paging
COW

Locality
Register allocation
Cache policy
Prefetching
Heterogeneity in processors

- Thread level parallelism
- Instruction level parallelism
Heterogeneity in processors

Thread level parallelism

Instruction level parallelism

Thread level parallelism

Instruction level parallelism

Heterogeneous Computing: OpenCL
CUDA by Example
Parallel Programming in OpenMP
Parallel Programming in OpenMP
MPI
Intel Xeon processor
Intel processor
Heterogeneity in programming languages

Performance

Programming productivity
Heterogeneity in programming languages

Performance

Programming productivity
Challenges in heterogeneous computing

Heterogeneous programs and data

Programming productivity (Safety and convenience)

Heterogeneous programming languages and models

Performance

Heterogeneous computing devices
Programming systems for heterogeneous computing
Programming systems for heterogeneous computing
Programming systems for heterogeneous computing
Programming systems for heterogeneous computing

- **Productivity**
  - Multilingual mixed-environment debugger [OOPSLA`09]
  - Dynamic FFI bug detector [PLDI`10]
  - Mutation-Based Fault localizer [ASE`15]

- **Performance**
  - Sampling-based call graph profiler [CC`07, TACO`16]
  - Adaptive GPU JIT compilation (in progress)
  - Distributed shared memory for GPUs and MICs (in progress)
Outline

- Introduction
- Programming systems
  - **Productivity**
    - Multilingual mixed-environment debugger
    - Dynamic FFI bug detector
    - Mutation-Based Fault localizer
    - JeannieCL Interface Language
  - **Performance**
    - Sampling-based call graph profiler
    - Adaptive GPU JIT compilation
    - Distributed shared memory for GPUs and MICs
- Conclusion
Multilingual mixed-environment debugger [OOPSLA`09]
Composing a Multilingual Debugger Using Single Language Debuggers

JDB

GDB

Managed Java

JNI

Native C/C++
Problem: GDB Does Not Work at a Java Breakpoint

- JDB
- GDB
- Managed Java
- JNI
- Native C/C++
Problem: GDB Does Not Work at a Java Breakpoint

How can I print a C variable at a Java breakpoint?
Unfortunately, GDB does not work at the Java breakpoint.
Our Solution: The Intermediate Agent Switches Debugger Context

We will wake up GDB at a Java breakpoint - Switching debugger context.
Mutation-Based Fault localizer [ASE`15]

Java heap

C heap

GC
root

Opaque
reference
Memory leak in C from Java

- GC root
- Reclaimed by GC
- Opaque reference
- Java heap
- C heap
Memory leak in C from Java

- **Cause in Java**
  - GC root

- **Fault in C**
  - Opaque reference

**Java heap**

**C heap**
Jinn: runtime monitoring for FFI bugs [PLDI`2010]

Standard libraries

Java
C/C++

Python
C/C++

Multilingual bindings

Java
C/C++

Python
C/C++

Ruby

Multilingual components

Java
C/C++

SQL
## Challenge: Multilingual Bugs Are Hard to Diagnose

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Traps and Pitfalls</td>
<td>131</td>
</tr>
<tr>
<td>10.1</td>
<td>Error Checking</td>
<td>131</td>
</tr>
<tr>
<td>10.2</td>
<td>Passing Invalid Arguments to JNI Functions</td>
<td>131</td>
</tr>
<tr>
<td>10.3</td>
<td>Confusing jclass with jobject</td>
<td>132</td>
</tr>
<tr>
<td>10.4</td>
<td>Truncating jboolean Arguments</td>
<td>132</td>
</tr>
<tr>
<td>10.5</td>
<td>Boundaries between Java Application and Native Code</td>
<td>133</td>
</tr>
<tr>
<td>10.6</td>
<td></td>
<td>134</td>
</tr>
<tr>
<td></td>
<td></td>
<td>135</td>
</tr>
<tr>
<td></td>
<td></td>
<td>136</td>
</tr>
<tr>
<td></td>
<td></td>
<td>137</td>
</tr>
<tr>
<td></td>
<td></td>
<td>138</td>
</tr>
<tr>
<td></td>
<td></td>
<td>139</td>
</tr>
<tr>
<td></td>
<td></td>
<td>140</td>
</tr>
<tr>
<td></td>
<td></td>
<td>141</td>
</tr>
</tbody>
</table>

FFI bugs are rampant

- 716 [Li & Tan ’09]
- 86 [Kondoh & Onodera ’08]
- 155 [Furr & Foster ’06]

However, a common pitfall is to extract an object from a list so almost any operation is potentially dangerous.
An Invalid JNI Reference in The GNOME Bug 576111

```c
void Bug_producer(
    JNIEnv *env,
    jobject lref){
    global = lref;
}
```
void Bug_producer(
JNIEnv *env,
jobject lref){
global = lref;
}

An Invalid JNI Reference in The GNOME Bug 576111
An Invalid JNI Reference in The GNOME Bug 576111

void Bug_consumer(
JNIEnv *env) {
  env->CallJ(global);
}

void Bug_producer(
JNIEnv *env, jobject lref) {
  global = lref;
}

void Bug_consumer(
JNIEnv *env) {
  env->CallJ(global);
}
The State Machine Detects the Bug

Before Acquire

Acquire

Acquired

Call:Java → C

void Bug_producer(
JNIEnv *env,
jobject lref){
global = lref;
}
The State Machine Detects the Bug

Before Acquire

Acquire

Acquired

release

Released

Call:Java → C

Return:C → Java

void Bug_producer( 
JNIEnv *env, 
jobject lref){
global = lref; 
}

37
The State Machine Detects the Bug

Before Acquire
  acquire
Acquired
  release
Released
  use
Error: Dangling

void Bug_producer( JNIEnv *env,
  jobject lref)
{
  global = lref;
}

void Bug_consumer( JNIEnv *env)
{
  env->callJ(global);
}
Synthesizing Dynamic Bug Detectors

State machine description

Synthesizer

JNI bug detector (Jinn)

Jinn [PLDI`2010]

Our synthesis approach applies to other FFIs including Python/C
Running Dynamic Bug Detectors

Our synthesis approach applies to other FFIs including Python/C
Mutation-based fault localization

- **Erroneous JNI program**
  - Mutated program 1 (mutant 1)
  - Mutated program 2 (mutant 2)
  - Mutated program 3 (mutant 3)

**Successful test case**
- Test coverage

**Failing test case**
- Coverage
  - Test coverage

- **Mutation-based fault localizer**
  - Fault cause location
  - Bug-fixing patch

**Success**

**Failure**
Introduction

Programming systems

- Productivity
  - Multilingual mixed-environment debugger
  - Dynamic FFI bug detector
  - Mutation-Based Fault localizer
  - JeannieCL Interface Language

- Performance
  - Sampling-based call graph profiler
  - Adaptive GPU JIT compilation
  - Distributed shared memory for GPUs and MICs

Conclusion
GPU/MIC Programming Interface

- CUDA and OpenCL
  - Annotate the offloaded functions
  - Annotate the shared variables

- Our approach
  - Innovate JVMs for transparent execution
  - Eliminate all annotation burdens
  - Adaptive automatic runtime system
    - Online profiling to decide which one to offload
    - Online JIT compilation
    - Distribute shared memory
Complexity of large object oriented programs
- Decompose the program into small methods
- Method boundary becomes performance-bottleneck

Dynamic interprocedural optimization
- Solve the method boundary problem
- Inlining and specialization vary the performance by factor of 2
- Dynamic call graph (DCG) is critical input!
Sampling-based call graph profiler [TACO`16, CC`07]

![Graph showing the trade-off between accuracy and overhead for different profiling methods. The graph includes points for full instrumentation [PLDI`82], timer-based sampling [OOPSLA`00], and sampling. The full instrumentation method has an accuracy of 100% with an overhead of 25%, while the timer-based sampling method has an accuracy of 5% with an overhead of 0%. The sampling method lies between these two, showing a trade-off between accuracy and overhead.]
Accuracy-overhead tradeoff

Accuracy (%) vs Overhead (%)

- Full instrumentation [PLDI’82]
- Arnold-Grove sampling [CC’05]
- Timer-based sampling [OOPSLA’00]

Ideal profiling
Accuracy-overhead tradeoff

Accuracy (%) vs. Overhead (%)

- Full instrumentation [PLDI’82]
- Constraint-based correction [CC’07]
- Adaptive correction [TACO’16]
- Arnold-Grove sampling [CC’05]
- Timer-based sampling [OOPSLA’00]
- Ideal profiling
- Current state-of-the-arts
Timer-based call graph sampling

Dynamic call graph
Timer-based call graph sampling

Call stack:

- Call b
- Call c
- Method a

DCG_Sample:

- 9→10 5
- 10→11 5
- 11→5→6 5
- 999→1000 500

Timer ticks:

- t
Applying flow conservation

\[ f(a \rightarrow c) = f(a \rightarrow c) \]

Flow conservation says something is wrong
Adaptive GPU JIT compiler (in progress)

Application program
- Java byte code program
- Adaptive computation offloader
- x86 JIT compiler
- PTX JIT compiler

GPU powered JVM
- x86 code
- PTX code

Machine code
- x86 CPU
- GPU

Heterogenous processors
- Execution profiles
- Software sampling
- Hardware sampling
Distributed shared memory (in progress)

- Java byte code
- Application program
- CPU JIT compiler
- Heterogeneous memory manager
- GPU JIT compiler
- Offloading controller
- X86 code
- Data
- PTX code
- Cached data
- PTX code
- Control flow
- Data flow
- Heterogeneous memory manager
- Heterogeneous processing environment
- Heterogeneous devices
- X86 CPU
- GPU
Conclusion

- Heterogeneous computing
  - The end of Dennard scaling
  - Heterogeneous devices, languages, programs, and data

- Programming systems
  - Productivity [ASE`15, PLDI`10, OOPSLA`09]
  - Performance [TACO`16, CC`07]

- Making heterogeneous computing easy
Thank You