A Concurrent Dynamic Analysis Framework for Multicore Hardware

Jungwoo Ha  Univ. Texas at Austin & USC/ISI-East
Matthew Arnold  IBM Research
Stephen M. Blackburn  Australian National University
Kathryn S. McKinley  The Univ. of Texas at Austin
Software & Hardware Trends

Past
- Faster Single Processor
  - frequency scaling
- Larger, More Capable Software
  - Managed Languages

Today
- More Cores
  - {multi, many} cores
- Scalable Software
  - scalable (applications + managed runtime)
Sequential managed programs

- Profiling
- Dynamic Analysis
- Compilation
- Garbage Collections
- Other Helper Threads
- ……
Steps towards scalability

Step 1. parallel application

- Core 0
- Core 1
- Core 2: Application
- Core 3: Threads
- Core 4
- Core 5
- Core 6
- Core 7

Time

Unused cores

Each thread has different length
Steps towards scalability

Step 2. parallel managed runtime

Managed runtime waits for all application threads to pause

J. Ha, M. Arnold, S. Blackburn, K. McKinley,
A Concurrent Dynamic Analysis Framework for Multicore Hardware, OOPSLA 2009
Steps towards scalability

Step 3. parallel & concurrent managed runtime

Managed runtime may be in application’s critical path perturbing the application’s performance a lot

J. Ha, M. Arnold, S. Blackburn, K. McKinley,
A Concurrent Dynamic Analysis Framework for Multicore Hardware, OOPSLA 2009
Steps towards scalability – Ideal model

Step 4. minimize perturbation

- Core 0
- Core 1
- Core 2
  - Application
  - Application
- Core 3
  - Threads
  - Threads
- Core 4
- Core 5
- Core 6
- Core 7

Instrumented code only produces source data

whole runtime task off the critical path
Steps towards scalability – Ideal model

**Step 4. minimize perturbation**

- Core 0
- Core 1
- Core 2: Application
- Core 3: Threads
- Core 4
- Core 5
- Core 6
- Core 7

Worst case is parallel & concurrent

---

*J. Ha, M. Arnold, S. Blackburn, K. McKinley,
A Concurrent Dynamic Analysis Framework for Multicore Hardware, OOPSLA 2009*
Outline

- Motivation
- Design
  - Sequential vs Concurrent Dynamic Analysis
  - Cache-friendly Asymmetric Buffering
- Evaluation
- Conclusion
Generic Sequential Analysis

- Instrumented code (== overhead)
- Data collection
- Analysis

- Difficult to optimize instrumented code
- Trade accuracy for overhead (sampling)
Generic Concurrent Analysis

- instrumented code (reduced overhead)
- data collection
- enqueue
- buffering
- dequeue
- analysis

- Lower overhead & higher accuracy
- Must deal with microarchitectural side-effects

Application (producer)

Analysis (consumer)
Side-effects to Avoid

- False & true sharing
- High latency memory operation
- Cache line ping-ponging

Core A → L1 → lower level cache(s) → L1 → Core B

Application (Producer) → Analysis (Consumer)
Cache-friendly Asymmetric Buffering

- Lock-free communication channel between application and analysis thread
- Cache-friendly buffering
  - Actively avoids micro-architectural side-effects
- Asymmetric buffering
  - enqueue operation
    - light-weight instrumentation
    - produces one record at time
  - dequeue operation
    - consumes one chunk (fraction of a buffer) at a time
Cache-friendly Asymmetric Buffering

Core A  
Application (Producer)

L1
lower level cache(s)
L1
Core B
Analysis (Consumer)

application writes here

application waits for application here.

analyzer reads here

- 16 slots on the buffer
- 4 chunks, 4 slots on each chunk
- L1 size == chunk size

J. Ha, M. Arnold, S. Blackburn, K. McKinley, 
A Concurrent Dynamic Analysis Framework for Multicore Hardware, OOPSLA 2009
Cache-friendly Asymmetric Buffering

- **Core A (Application (Producer))**
  - L1
    - 0
    - 1
    - 2
    - 3
  - Lower level cache(s)
  - L1

- **Core B (Analysis (Consumer))**

---

- **Delay consumer dequeue operation until cache line is flushed**
  - 2 chunks away (smiley location)

- **Analyzer operates one chunk at a time**
  - chunk_size > L1 size
  - In practice, chunk_size >= 2 * L1 works well.

---

*J. Ha, M. Arnold, S. Blackburn, K. McKinley, A Concurrent Dynamic Analysis Framework for Multicore Hardware, OOPSLA 2009*
Cache-friendly Asymmetric Buffering

- **Core A**
  - Application (Producer)

- **L1**

- **Lower level cache(s)**
  - 4: 0
  - 5: 1
  - 6: 2
  - 7: 3

- **Core B**
  - Analysis (Consumer)

**Application blocks only when buffer is full**
- Waiting until two more chunks are available

---

*J. Ha, M. Arnold, S. Blackburn, K. McKinley,*
*A Concurrent Dynamic Analysis Framework for Multicore Hardware, OOPSLA 2009*
Producer (sees buffer)

while (*bufptr != 0) {
    if (*bufptr == MAGIC)
        bufptr = buffer;
    if (*bufptr != 0)
        block();
}
*bufptr++ = data;

Consumer (sees chunk)

while (app_is_running) {
    index = index_of(chunk_num+2);
    while (buffer[index] == 0)
        spin_or_sleep();
    consume(chunk_num);
    chunk_num = NEXT(chunk_num);
}

- producer may spin on bufptr, while consumer may spin on buffer
- Producer code common case is 6 instructions in x86.
Framework Provides ...

- Cache-friendly Asymmetric Buffering (CAB)
  - Minimizes micro-architectural side-effects
  - Quickly offloads event data from application’s critical path
- Configurable parameters for optimization
  - buffer size & chunk size
- Various collection mode
  - Exhaustive mode
  - Sampling mode
- Works on various threading model
  - N:M (green) threading model
  - native threading model

J. Ha, M. Arnold, S. Blackburn, K. McKinley,
A Concurrent Dynamic Analysis Framework for Multicore Hardware, OOPSLA 2009
Outline

- Motivation
- Design
  - Sequential vs Concurrent dynamic analysis
  - Cache-friendly Asymmetric Buffering
- Evaluation
- Conclusion
Evaluation

3 different Intel processors

Pentium 4 w/ hyperthreading
Core 2 Quad
Core i7

J. Ha, M. Arnold, S. Blackburn, K. McKinley,
A Concurrent Dynamic Analysis Framework for Multicore Hardware, OOPSLA 2009
Evaluation

- Jikes RVM (2 different threading models)
  - N:M threading (Jikes RVM 2.9.2)
  - Native threading (Jikes RVM 3.0.1)
- Reference Dynamic Analysis Implementation
  - Method counting
  - Call graph
  - Call tree profiling
  - Path profiling
  - Cache simulator using load/store events
- Benchmarks
  - DaCapo, SPEC JVM 98 benchmark suites
- Parameters
  - buffer size = 2MB, chunk size = 128KB
Call Graph Profiling

- **Instrumentation Overhead** – *Bar 1*
- Bar 1 – Collect event data and write into a single word. No analysis thread
Enqueueing Overhead – (Bar2 - Bar 1)

Bar2 – Collect event data and write into the buffer. No analysis thread
Call Graph Profiling

Communication Overhead – (Bar3 – Bar 2)

Bar3 – Analysis thread dequeues and write it into a single word.
Call Graph Profiling

- **Analysis (data processing) Overhead** – (Bar 5 – Bar 1)
- Bar4 – Concurrent Analysis
- Bar5 – Sequential Analysis

J. Ha, M. Arnold, S. Blackburn, K. McKinley, 
*A Concurrent Dynamic Analysis Framework for Multicore Hardware*, OOPSLA 2009
Call Graph Profiling

- **Overhead reduction with Concurrent Analysis** – (Bar 5 – Bar 4)
- Bar4 – Concurrent Analysis
- Bar5 – Sequential Analysis
Path Profiling

- **Overhead reduction with Concurrent Analysis** – (Bar 5 – Bar 4)
- More data & computation than call graph

*J. Ha, M. Arnold, S. Blackburn, K. McKinley, A Concurrent Dynamic Analysis Framework for Multicore Hardware, OOPSLA 2009*
Path Profiling – Multithreaded Benchmarks

Overhead (%)

- Instrumentation
- Enqueue
- Enqueue+Dequeue
- Concurrent (N:M)
- Sequential

- Core 2
- Multi-threaded benchmarks

J. Ha, M. Arnold, S. Blackburn, K. McKinley,
A Concurrent Dynamic Analysis Framework for Multicore Hardware, OOPSLA 2009
Related Work

- Concurrent Lock-free Queue
  - FastForward – single-producer & single-consumer [Giacomoni et al. 09]

- Concurrent analysis for specific clients
  - PiPA – cache simulator [Zhao et al. 08]

- Shadow process approach
  - Shadow profiling [Moseley et al. 07]
  - SuperPin [Wallace et al. 07]
Future Work

- Per-processor buffering on native threading model

- Self-tuning parameters
Conclusions

- We introduced concurrent dynamic analysis framework that helps implementing concurrent analysis.

- CAB efficiently transfers analysis data from application to analysis thread avoiding microarchitectural side-effects.

- Our framework efficiently utilizes extra cycles to perform dynamic analysis concurrently.
THANK YOU

Jungwoo Ha (jha@isi.edu)
>97% accurate at 5% sampling rate.

Accuracy drops when the profiler is slower than the application.
Sampling Overhead – Path profiling

- Overhead reduction from exhaustive to 5% sampling
  - Antlr 117% → 49%
  - Hsqldb 124% → 44%
  - Luindex 95% → 38%
A Concurrent Dynamic Analysis Framework for Multicore Hardware, OOPSLA 2009