Predictable Timing Analysis of x86 Multicores using High-Level Parallel Patterns

Kevin Hammond, Susmit Sarkar and Chris Brown
University of St Andrews, UK

T: @paraphrase_fp7
E: kh@cs.st-andrews.ac.uk
W: http://www.paraphrase-ict.eu
Motivation

• No future system will be single-core
  - parallel programming will be essential

• It’s not just about performance
  - it’s also about energy usage

• If we don’t solve the multicore challenge, then no other advances will matter!

• We need to produce predictable timing models for widely used multicores (e.g. x86, ARM)
Even Mobile Phones are Multicore!
Current Parallel Methodologies

- Applications programmers must be systems programmers
  - insufficient assistance with abstraction
  - too much complexity to manage
- Difficult/impossible to scale, unless the problem is simple
- Difficult/impossible to change fundamentals
  - scheduling
  - task structure
  - migration
- Many approaches provide libraries
  - they need to provide abstractions
Thinking Parallel

• Fundamentally, programmers must learn to “think parallel”
  - this requires new *high-level* programming constructs
    - perhaps dealing with large numbers of threads

• You cannot program effectively while worrying about deadlocks etc.
  - they must be eliminated from the design!

• You cannot program effectively while fiddling with communication etc.
  - this needs to be packaged/abstracted!

• You cannot program effectively without performance information
  - this needs to be included as part of the design!
The ParaPhrase Approach

Original Code

Pattern Library

Parallel Code

Erlang

C/C++

Haskell

Costing/Profiling

Refactoring

Mellanox Infiniband

Nvidia Tesla

AMD Opteron

AMD Opteron

Intel Core

Intel Core

Intel Xeon Phi

Nvidia GPU

Nvidia GPU

Intel GPU

Intel GPU
Components and Abstraction

- **Components give some of the advantages of functional programming**
  - clean abstraction
  - pure computations, easily scheduled
  - dependencies can be exposed

- **Hygiene/discipline is necessary**
  - no unwanted state leakage
    (e.g. in terms of implicit shared memory state)
The ParaPhrase Approach

• Start bottom-up
  – identify (strongly hygienic) COMPONENTS
  – using semi-automated refactoring

• Think about the PATTERN of parallelism
  – e.g. map(reduce), task farm, parallel search, parallel completion, ...

• STRUCTURE the components into a parallel program
  – turn the patterns into concrete (skeleton) code
  – Take performance, energy etc. into account (multi-objective optimisation)
  – also using refactoring

• RESTRUCTURE/TUNE if necessary! (also using refactoring)

both legacy and new programs
Some Common Parallel Patterns

Generally, we need to nest/combine patterns in arbitrary ways.
Skeletions

- Skeletons are implementations of parallel patterns
- A skeleton is a template
  - pluggable higher-order functions
  - can be instantiated with concrete worker functions
- Skeletons avoid deadlock, race conditions
  - communication is implicit and structured


Parallel Pipeline Skeleton

- Each stage of the pipeline can be executed in parallel
- The input and output are streams
- Each stage is itself an instance of a pattern (Skel)

\[
\text{skel:do}([\{\text{pipe}, [\text{Skel}_1, \text{Skel}_2, \ldots, \text{Skel}_n]\}], \text{Inputs}).
\]
Parallel Task Farm Skeleton

- Each worker is executed in parallel
- A bit like a 1-stage pipeline

```
skel:do([{farm, Skel, M}], Inputs).
```
Example Parallel Structure

Sequential

for each image, i.
process(read i)

Parallel

{pipe, {farm, {func, read}, m},
{farm, {func, process}, n}}
Composing Skeletons

- Queues link skeletons

```
{pipe, {farm, {func, read}, m},
 {farm, {func, process}, n}}
```
x86 Multicore Cache Design

- Each core has
  - a local write-back cache
  - a FIFO-ordered **write buffer**

- A core may run many threads

- Cores share
  - level 2 (and 3) cache
  - global memory
Sequential Consistency (SC)

```
int x = 0, y = 0;
...

{ // thread 1
  x = 1;
  return(x+y);
}

{ // thread 2
  y = 2;
  return(x+y);
}
```

- Memory accesses are effectively interleaved
  - as if run by a single processor

- Either
  - both threads return 3
  - thread 1 returns 1, thread 2 returns 3
  - thread 1 returns 3, thread 2 returns 2

- Not
  - thread 1 returns 0,
  - thread 2 returns 0
x86 Total Store Order (TSO)

- On a multicore, SC can be inefficient
- Intel uses a weaker (relaxed memory) consistency model
  - Total store order (TSO) guarantees that the order that \textbf{Writes} are seen by a location is the same as the order they were issued
- ARM uses an even weaker consistency model
Basic TSO Rules

• The basic rules are:
  (1) **Reads** are not reordered with other **Reads**.
  (2) **Writes** are not reordered with older **Reads**.
  (3) **Writes** are not reordered with other **Writes**.
  (4) **Reads** may be reordered with older **Writes** to different memory locations but not with older **Writes** to the same memory location.

• An **Exchange** is treated as an indivisible Read/Write pair to a specific memory location.

• A **Fence** is treated as both a Read and Write to all memory locations, except that no actual memory transfer occurs.
Simple Spin Lock Implementation

```c
void lock( volatile char *lockcell ) {
    char old_value ;
    do {
        old_value = exchange(lockcell,1);
    } while ( 1 == old_value );
}

void unlock( volatile char *lockcell ) {
    *lockcell = 0 ;
}
```
x86 Assembly code for spin lock

```assembly
lockr:
    push ebp ; Start new stack frame
    mov ebp, esp
    mov ebx, [ebp+8] ; Get address of lock cell

trylock:
    mov eax, 1 ; Set EAX register to 1 (locked)
    xchg eax, [ebx] ; Exchange EAX and lock cell
    test eax, eax ; Test whether the cell is already locked
    jnz trylock ; Retry the lock if so
    pop ebp ; revert stack frame
    ret ; The lock has been acquired

unlockr:
    push ebp ; Start new stack frame
    mov ebp, esp
    mov ebx, [ebp+8] ; Get address of lock cell
    mov eax, 0 ; Set EAX register to 0 (unlocked)
    mov [ebx], eax ; Release the lock

    pop ebp ; revert stack frame
    ret ; The lock has been released.
```
Simple Queue using spin lock

```c
Value qget(Queue q) {
  Value v;
  do {
    lock(&q.lock_cell);
    if (qempty(q))
      break;
    unlock(&q.lock_cell);
  } while (1);
  /* lock is held */
  v = front(q);
  unlock(&q.lock_cell);
  return(v);
}

void qput(Queue q, Value v) {
  lock(&q.lock_cell);
  addtoq(q,v);
  unlock(&q.lock_cell);
}
```

We have used HOL to prove that this is sound wrt the TSO relaxed-memory model.
Simple Timing Model

- The worst-case costs if \( n \) threads contend a lock are

\[
T_{qput} = n \cdot T_{Exchange} + T_{Write} + T_{Write}
\]

\[
T_{qget} = n \cdot T_{Exchange} + T_{Read} + 2T_{Write}
\]
Timing Model for a Farm

- The amortised average cost for each farm operation is

\[ T_{qget} + T_f + T_{qput} \]

which simplifies to:

\[ 2 \cdot (n + 1) \cdot T_{Exchange} + 5 \cdot T_{Write} + T_{Read} + T_f \]
Timing Model for a Pipeline

• If the first stage dominates (function $f$), its cost is

$$T_{qget} + T_f + T_{qput}$$

which simplifies to:

$$2 \cdot (|f| + 1) \cdot T_{Exchange} + 5 \cdot T_{Write} + T_f$$

• The total cost for both stages is therefore:

$$2 \cdot (|f| + 1) \cdot T_{Exchange} + 5 \cdot T_{Write} + T_f + T_g$$

or, if the second stage dominates (function $g$)

$$T_f + (2 \cdot |g| + |f| + 1) \cdot T_{Exchange} + 5 \cdot T_{Write} + T_g$$
Including Store-Buffer Flushing

- The cost of an exchange depends on items to be flushed, $b$
  \[ T_{\text{Exchange}} = b \cdot T_{FL} + T_{\text{JustX}} \]
- The cost of a spin-lock on $t$ contending threads is
  \[ b \cdot T_{FL} + t \cdot T_{\text{JustX}} \]
- The costs of queue operations change slightly
  \[ T_{\text{qput}} = b \cdot n \cdot T_{FL} + n \cdot T_{\text{JustX}} + T_{\text{Write}} + T_{\text{Write}} \]
  \[ T_{\text{qget}} = b \cdot n \cdot T_{FL} + n \cdot T_{\text{JustX}} + T_{\text{Read}} + 2 \cdot T_{\text{Write}} \]
- The cost of a farm is:
  \[ 2 \cdot (n + 1) \cdot b \cdot T_{FL} + (n + 1) \cdot T_{\text{JustX}} + 4T_{\text{Write}} + T_{\text{Read}} + T_{f} \]
Performance Predictions
(Image Convolution, 1024x1024)

24 core machine at Uni. Pisa
2xAMD Opteron 6176. 800 Mhz
32GB RAM
1 x NVidia Tesla C2050 GPU
Performance Predictions
(Image Convolution, 2048x2048)
Performance Predictions
(Image Convolution, 2048x2048)

64-core machine at Uni. St Andrews
8xAMD Opteron 6376. 2.3Ghz
32GB RAM
Performance Predictions
(Matrix Multiplication etc)
Comparison with OpenMP

Dashed lines are OpenMP
Combining CPUs and GPUs

- **Machine Learning chooses**
  - best combination of patterns
  - CPU/GPU allocations

- **Excellent Results**
  - within 5% of optimal
  - > 40x speedup over sequential CPU
Lock-Free Queue Implementation

```c
-- Value qgetlf(Queue q) {
    NodePtr first;
    do {
        if (qempty(q))
            continue;
        first = q->first;
        if (first == NULL)
            continue;
    } while (!dcas(q->first, first, first->next));
    return(first->value);
}
```

- This uses a double compare-and-swap variant of Exchange
  - atomically swaps two values
  - allows us to avoid ABA errors by including a count field
Lock-Free Queue Implementation

```java
void qputlf(Queue q, Value v) {
    Node n = new Node(v);       // new queue node
    NodePtr np = new NodePtr(n); // queue pointer
    Queue last;
    do {
        last = q->last;
        np.count = last.count+1;
    } while(!dcas(q->last, last, np));
}
```

- **At the pattern level, this is plug-replaceable with a lock**
  - The cost model needs to change but most details are the same
  - All proof is the same above the lock-free level
Comparison of Development Times

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Man.Time</th>
<th>Refac. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolution</td>
<td>3 days</td>
<td>3 hours</td>
</tr>
<tr>
<td>Ant Colony</td>
<td>1 day</td>
<td>1 hour</td>
</tr>
<tr>
<td>BasicN2</td>
<td>5 days</td>
<td>5 hours</td>
</tr>
<tr>
<td>Graphical Lasso</td>
<td>15 hours</td>
<td>2 hours</td>
</tr>
</tbody>
</table>

Figure 3: Comparison of manual vs. refactored times.
Conclusions

• High-level Patterns help structure parallel computations
  - avoid deadlock, race conditions etc (formal proof in paper!)
  - reduce development time by an order of magnitude
  - allow us to construct predictable cost models

• Cost model for x86 constructed from first principles
  - Predictable timings for x86 (provably correct from TSO semantics)
  - Highly Accurate
  - All previous formal models have been for much simpler memory models (e.g. PPC)

• Proved to be deadlock-free

• Applicable to energy as well as time
Funded by

• ParaPhrase (EU FP7), Patterns for heterogeneous multicore, €4.2M, 2011-2014

• SCIEnce (EU FP6), Grid/Cloud/Multicore coordination
  • €3.2M, 2005-2012

• Advance (EU FP7), Multicore streaming
  • €2.7M, 2010-2013

• HPC-GAP (EPSRC), Legacy system on thousands of cores
  • £1.6M, 2010-2014

• Islay (EPSRC), Real-time FPGA streaming implementation
  • £1.4M, 2008-2011

• TACLE: European Cost Action on
  • €300K, 2012-2015
Some of our Industrial Connections

Mellanox Inc.
Erlang Solutions Ltd
SAP GmbH, Karlsruhe
BAe Systems
Selex Galileo
Biold GmbH, Stuttgart
Philips Healthcare
Software Competence Centre, Hagenberg
Microsoft Research
Well-Typed LLC
ICT COST Action IC1202

Timing Analysis on Code-Level (TACLE)

Descriptions are provided by the Actions directly via e-COST.

Embedded systems increasingly permeate our daily lives. Many of those systems are business- or safety-critical, with strict timing requirements. Code-level timing analysis (used to analyse software running on some given hardware w.r.t. its timing properties) is an indispensable technique for ascertaining whether or not those requirements are met. However, recent developments in hardware, especially multi-core processors, and in software organisation render
THANK YOU!

http://www.paraphrase-ict.eu

http://www.project-advance.eu

@paraphrase_fp7