An Automatic Runtime DOALL Loop Parallelisation Optimization for Java

Dr. Ian Rogers, Jisheng Zhao,
Dr. Chris Kirkham, Prof. Ian Watson
The Advanced Processor Technologies group
http://www.cs.manchester.ac.uk/apt
Presentation outline:

- Motivation
  - simultaneous multithreading, chip multiprocessor architectures
  - the JAMAICA architecture
  - work distribution
  - virtualization
- Annotated Loop Structure Trees
- Null and bound check elimination
- Parallelisation optimisation
- Performance analysis
  - SpecJVM 98
  - simple kernel
- Future work
- Summary
SMT and CMP Architectures

- Simultaneous MultiThreading (SMT):
  - performance gap between processor and memory is growing
  - threads can be scheduled on cache misses to hide memory access time
- Chip MultiProcessors (CMP):
  - instruction level parallelism reaching limits
  - reduce design complexity
  - local clocks aid clock distribution
- Threaded code necessary to expose parallelism
- New mechanisms to help expose threaded parallelism
  - thread scheduling and work distribution
  - speculative threading (transactional commit mechanism)
- This work is a first step into a runtime support system
Overview of the JAMAICA architecture
Some more detail

- Alpha based instruction set
- 5 stage MIPS based pipeline (without TLB stages)
- Heap allocated registers and context management
- Token ring interface

Combining the strengths of UMIST and The Victoria University of Manchester
Work distribution

- Idle threads distribute tokens on a token ring bus
- Executing context on a core requests to ship work to an idle context or core and context
- Taking a token from ring grants the use of a particular context
- Shipping of work between cores occurs over data bus
- Gives lightweight thread creation
- When token is redistributed, work has been completed
- Thread unit monitors for completion of forked work
Virtualization

- Platform independence
  - Operating system virtualization
    - Run multiple operating systems simultaneously on virtualized hardware
  - Application virtualization
    - Standard application formats such as ELF can run on a multitude of operating systems as binary format and system call interface are standardized.
    - Wine allows windows applications to run on FreeBSD, Linux and Solaris
  - Instruction set virtualization
    - Dynamic binary translators (see presentation in tomorrows PLOS workshop)
- Hardware flexibility
  - Transmeta - 4-way VLIW TM3000 and TM5000 processors, 8-way VLIW TM8000 processor all run IA32 code
- New compiler optimizations …
Software support for the JAMAICA architecture

- Tools
  - C compiler – based on Princeton’s LCC
  - jtrans – Java class file to assembler
  - javar – modified to generate jtrans parallel constructs
  - sim-idbg – interactive debugger and simulator in C
  - SIMPA – threaded, interactive, cycle accurate and fast simulator in Java
  - Jikes RVM – JAMAICA back-end and runtime
The Jikes RVM

- JVM written in Java
- Support for IA32, PowerPC and JAMAICA
- Baseline (quick) and optimizing compilers
- Adaptive optimization and feedback system
- Extended array SSA form sub-stages in HIR and LIR optimization
Annotated Loop Structure Trees

- Loop Predecessor
- Loop Header
  \[
  \text{iterator1} = \phi(\text{initial\_value}, \text{iterator2})
  \]
- Loop Body
- Loop Exit
  \[
  \text{iterator2} = \text{iterator1} + \text{stride}
  \]
  \[
  \text{if (iterator < terminal\_value) goto}
  \]
- Loop Successor

Loop blocks may be separate or combined

Stride and compare operators can vary from those shown, as can the placement of the iterator2 computation.
Null and bound check elimination

- ABCD analysis eliminates checks when the values of the array length and non-nullness are known
- Length and non-nullness are known following a test or after an array is created
- Analysis of spec benchmarks showed ABCD wasn’t enabling loops to be parallelisable
- Annotated LST used to duplicate loop body and create one without tests and one with, with explicit tests beforehand

```java
for (int i = fromIndex; i < toIndex; i++) {
    g1 = null_check a;
    g2 = bounds_check a, g1;
    g3 = guard_combine (g1, g2);
    a[i] = val, g3;
}
```
Duplicated loops, one without exceptions

Combining the strengths of UMIST and The Victoria University of Manchester
Performance of Annotated LST Optimizations

- 201_compress
- 202_jess
- 205_raytrace
- 209_db
- 213_javac
- 222_mpegaudio
- 227_mtrt
- 208_jack

Performance Alteration (%)

- constant loop unrolling
- loop unrolling
- null/boundcheck elimination
Loop Parallelisation Optimisation

```java
class Jamaica_LoopWorker_X extends Jamaica_LoopWorker {
    int[] loopInvariantA;
    int[] loopInvariantB;

    loopWorker(int iterator) {
        [...]
        X = loopInvariantA[iterator];
        Y = loopInvariantB[iterator];
        [...]
    }
}
```

Jamaica_loopWorker_X lw =
new Jamaica_loopWorker_X();

lw.loopInvariantA = A;
lw.loopInvariantB = B;

JamaicaThreads.runLoopWorkers
(lw, initial_value, terminal_value);

Copy loop invariants to worker

Create, forks and joins parallel threads. Each thread runs the loopWorker multiple times

loopWorker is a modified version of the loop body
### Parallelised SpecJVM Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Parallelisation speed-up&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Overhead</th>
<th>Normal benchmark execution time</th>
<th>Executed parallel loop bodies</th>
</tr>
</thead>
<tbody>
<tr>
<td>_201_compress</td>
<td>1.6%</td>
<td>10.5s</td>
<td>7.1s</td>
<td>3500</td>
</tr>
<tr>
<td>_202_jess</td>
<td>0.7%</td>
<td>3.8s</td>
<td>3.5s</td>
<td>1500</td>
</tr>
<tr>
<td>_205_raytrace</td>
<td>3.7%</td>
<td>3.0s</td>
<td>4.3s</td>
<td>2400</td>
</tr>
<tr>
<td>_209_db</td>
<td>0.9%</td>
<td>8.4s</td>
<td>11.5s</td>
<td>4500</td>
</tr>
<tr>
<td>_213_javac</td>
<td>2.1%</td>
<td>0.2s</td>
<td>5.1s</td>
<td>1200</td>
</tr>
<tr>
<td>_222_mpegaudio</td>
<td>2.7%</td>
<td>11.9s</td>
<td>7.2s</td>
<td>12500</td>
</tr>
<tr>
<td>_227_mtrt</td>
<td>2.6%</td>
<td>1.7s</td>
<td>5.0s</td>
<td>1200</td>
</tr>
<tr>
<td>_228_jack</td>
<td>1.1%</td>
<td>1.3s</td>
<td>5.6s</td>
<td>1800</td>
</tr>
</tbody>
</table>

**Average speed-up of 1.9%**

<sup>1</sup>This is the performance speed-up excluding overheads introduced by creating threads and performing the optimisation.
Parallelised SpecJVM Performance
Simple kernel performance

- Simple test to see if optimisation can parallelise and get performance from simple case
- Performs no useful work 😊
- Achieved 79% speed-up on dual CPU Intel

```java
int size = 3000;
double[] matrix1 = new double[size];
double[] matrix2 = new double[size];
double[] result = new double[size];
for (int i = 1; i <= 500; i ++)
    for (int p = 0; p < size; p ++)
        matrix1[p] = p * p / i;
    matrix2[p] = p * (p + 1) / i;
result[p] = (i * p + 1) / i;
```
Future work

- Speculative execution
  - Range of speculative and non-speculative execution states
    - tree rooted at non-speculative state with branches for every spawned speculative context
    - speculative contexts may spawn more speculative contexts
  - If speculation goes wrong squash speculative state
    - throw away values in cache or a buffer
  - Detect speculation problems:
    - in software: when a value isn’t that expected explicitly squash
    - in hardware: when an address is loaded by a speculative context, ensure that stores to the same address from a less speculative context cause a squash
  - Problems with creating speculative threads and avoiding excessive squashing
  - Mechanism may aid virtual machines, e.g. handling of unaligned memory accesses
Future work

- Loop parallelisation can recognize more loops if loops with break out paths are including in analysis
- Parallelisation can work for these loops with more speculative threads being squashed if a break-out path is taken
Summary

- We have presented a series of runtime optimisations designed to increase the number of parallel threads for next generation CPUs.
- Threads are light-weight and may comprise just 1000s of instructions.
- Our optimisation doesn’t work on current CPUs with the current threading model (upto 2.48 times slow-down).
- Performance improvements on a standard benchmark suite are modest (1.9% on SpecJVM ignoring threading costs).
- Future hardware support for light-weight and speculative threading should improve the situation:
  - cheaper to create threads (e.g. JAMAICA)
  - possible to create more threads.
- We have a portable infrastructure for virtualization of the CPU, this work includes work on a Java oriented operating system and legacy code execution environment.
Thanks!

• … and any questions?