PROFILE-DIRECTED SEMI-AUTOMATIC PARALLELISATION

Björn Franke
Institute for Computing Systems Architecture
University of Edinburgh

Erlangen, 11 July 2011
Overview

• Motivation
• Part 1: Profile Directed Dependence Analysis
• Part 2: Machine Learning Based Mapping
• Part 3: Extraction of Pipeline Parallelism
• Conclusions
Motivation

• Multi-cores are ubiquitous
  • Try buying a single-core mobile phone, netbook, PC, or whatsoever!

• Legacy code base of sequential applications
  • Writing parallel applications is hard!

• No single parallel machine model
  • Different parallel programs for different parallel machines
  • Parallelisation is not a one-off activity: Need to parallelise each application for each new platform again
Motivation

- Multi-cores are ubiquitous
- Try buying a single-core mobile phone, netbook, etc.

**Tool Support for Parallelisation Increases Programmer Efficiency, Reduces Time-to-Market, Reduces Number of Bugs, Secures Software Investments etc.**

**BUT...**

- Parallelisation is *not* a one-off activity: Need to parallelise each application for each new platform again
Complete Failure of Auto-Parallelisation Despite >30 Years of Intensive Research!

NAS PB 2.3 OMP-C and SPEC CFP2000
2 Quad-cores (8 cores in total) Intel Xeon X5450 @ 3.00GHz
Intel icc 10.1 -O2 -xT -axT -ipo
Observations

• Static Dependence Analysis Doesn’t Work
  ➔ Part 1: Profile Directed Dependence Analysis

• Mapping of Parallelism is Really Hard
  ➔ Part 2: Machine Learning Based Mapping

• Parallelising FOR Loops is Not Enough
  ➔ Part 3: Extraction of Pipeline Parallelism
PART 1:
PROFILE DIRECTED
DEPENDENCE ANALYSIS
Motivating Example

\textbf{SPEC equake (~75\% of total exec. time)}

\begin{verbatim}
for (i = 0; i < nodes; i++) {
    Anext = Aindex[i];
    Alast = Aindex[i + 1];

    sum0 = A[Anext][0][0]*v[i][0] +
           A[Anext][0][1]*v[i][1] +
           A[Anext][0][2]*v[i][2];
    sum1 = ... 
    Anext++;
    while (Anext < Alast) {
        col = Acol[Anext];

        sum0 += A[Anext][0][0]*v[col][0] +
                A[Anext][0][1]*v[col][1] +
                A[Anext][0][2]*v[col][2];
        sum1 += ...
        w[col][0] += A[Anext][0][0]*v[i][0] +
                    A[Anext][1][0]*v[i][1] +
                    A[Anext][2][0]*v[i][2];
        w[col][1] += ...
        Anext++;
    }
    w[i][0] += sum0;
    w[i][1] += ...
}
\end{verbatim}

- Static analysis fails to detect any parallelism
- Problems:
  - indirect array accesses
  - compl. array reductions
  - variable iteration count
  - pointer aliasing
  - dynamic memory allocation
- But: Loop is parallel for all legal data inputs!
Profile Driven Parallelism Detection

- Use of profiling to capture data and control flow
  - Directly observe dependences ➔ accuracy
- But: Need to solve two important problems
  - No general correctness proof
    - May have missed dependences
  - Assisted user validation ➔ semi-automatic
- Use low-level profiling information in compiler?
  - Instrumentation of intermediate representation
  - No actual ISA idiosyncracies
Approach

- Instrument using high-level intermediate representation
  - Access to source-level information (memory accesses, loops, induction/reduction vars)
- Avoids ISA obfuscation
- Execute natively
  - Generates data and control flow traces
- Straightforward back-annotation
  - References to symbol table, IR nodes
- Combination with conventional static analyses
PART 2:
MACHINE LEARNING BASED MAPPING
Motivating Example

NAS CG

```c
#pragma omp for reduction(+:sum) private(d)
for (j=1; j <= lastcol-firstcol-1; j++) {
    d = x[j] - r[j];
    sum = sum + d * d;
}
```

OpenMP

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>Cell BE</th>
<th>Intel Xeon</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC</td>
<td>slowdown</td>
<td>2.3x</td>
</tr>
<tr>
<td>DYNAMIC</td>
<td>slowdown</td>
<td>slowdown</td>
</tr>
</tbody>
</table>
Problem Statement

Parallel loops
Problem Statement

Parallel loops

Statically provable parallel
Problem Statement

- **Unexploited**
- Statically provable parallel
Problem Statement

Unexploited

Statically provable parallel

Profitable
Holistic Approach: Detection & Mapping

Sequential code in C

Profiling-driven analysis

Code with OpenMP annotations

Machine-Learning based mapper

Code with profitable loops
Conventional Mapping Heuristics

Static not suitable for separation.

Linear models not adequate either!
Machine Learning Based Mapping

- **Off-line** learning
- Predict using **smallest** input

<table>
<thead>
<tr>
<th>Features</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>IR instruction count</td>
<td>Data access count</td>
</tr>
<tr>
<td></td>
<td>IR Load/Store count</td>
<td>Instruction count</td>
</tr>
<tr>
<td></td>
<td>IR Branch count</td>
<td>Branch count</td>
</tr>
<tr>
<td></td>
<td>Loop iteration count</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:
- New program
- Feature extraction
- Smallest input
- Trained model
- Profitable or not
- Scheduling policy
Support Vector Machine (SVM)

- Decide (i) profitability, (ii) loop scheduling
- Hyperplanes in transformed higher-dimensional space
- Non-linear & multi-class extensions
Experimental Evaluation

• 2 sets of applications
  • NAS Parallel Benchmarks 2.3
  • SPEC FP2000
• Sequential code in C
• Manually parallelized using OpenMP by expert programmers
• Use of multiple input datasets
Parallelism Coverage

- Profile-driven: almost **no lost opportunities**

<table>
<thead>
<tr>
<th>Application</th>
<th>FP</th>
<th>FN</th>
</tr>
</thead>
<tbody>
<tr>
<td>bt</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cg</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ep</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ft</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>is</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lu</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mg</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>sp</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>equake</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>art</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ammp</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Parallelism Coverage

- **MG**: 3 loops never execute for all inputs
- **ammp**: critical loops require reshaping & locking

<table>
<thead>
<tr>
<th>Application</th>
<th>FP</th>
<th>FN</th>
</tr>
</thead>
<tbody>
<tr>
<td>bt</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cg</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ep</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ft</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>is</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lu</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mg</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>sp</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>equake</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>art</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ammp</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Parallelism Coverage

- ICC finds many parallel loops

<table>
<thead>
<tr>
<th>Application</th>
<th>ICC #loops(%cov)</th>
<th>Profile-driven #loops(%cov)</th>
<th>Manual #loops(%cov)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bt</td>
<td>72 (18.6%)</td>
<td>205 (99.9%)</td>
<td>54 (99.9%)</td>
</tr>
<tr>
<td>cg</td>
<td>16 (1.10%)</td>
<td>28 (93.1%)</td>
<td>22 (93.1%)</td>
</tr>
<tr>
<td>ep</td>
<td>6 (&lt;1%)</td>
<td>8 (99.9%)</td>
<td>1 (99.9%)</td>
</tr>
<tr>
<td>ft</td>
<td>3 (&lt;1%)</td>
<td>37 (88.2%)</td>
<td>6 (88.2%)</td>
</tr>
<tr>
<td>is</td>
<td>8 (29.4%)</td>
<td>9 (28.5%)</td>
<td>1 (27.3%)</td>
</tr>
<tr>
<td>lu</td>
<td>88 (65.9%)</td>
<td>54 (99.7%)</td>
<td>29 (81.5%)</td>
</tr>
<tr>
<td>mg</td>
<td>9 (4.70%)</td>
<td>48 (77.7%)</td>
<td>12 (77.7%)</td>
</tr>
<tr>
<td>sp</td>
<td>178 (88.0%)</td>
<td>287 (99.6%)</td>
<td>70 (61.8%)</td>
</tr>
<tr>
<td>equake</td>
<td>29 (23.8%)</td>
<td>69 (98.1%)</td>
<td>11 (98.0%)</td>
</tr>
<tr>
<td>art</td>
<td>16 (30.0%)</td>
<td>31 (85.6%)</td>
<td>5 (65.0%)</td>
</tr>
<tr>
<td>ammp</td>
<td>43 (&lt;1%)</td>
<td>21 (1.40%)</td>
<td>7 (84.4%)</td>
</tr>
</tbody>
</table>
Parallelism Coverage

- BUT: low sequential time coverage
- ICC: majority of loops **too short** to be profitable

<table>
<thead>
<tr>
<th>Application</th>
<th>icc #loops(%cov)</th>
<th>Profile-driven #loops(%cov)</th>
<th>Manual #loops(%cov)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bt</td>
<td>72 (18.6%)</td>
<td>205 (99.9%)</td>
<td>54 (99.9%)</td>
</tr>
<tr>
<td>cg</td>
<td>16 (1.10%)</td>
<td>28 (93.1%)</td>
<td>22 (93.1%)</td>
</tr>
<tr>
<td>ep</td>
<td>6 (&lt;1%)</td>
<td>8 (99.9%)</td>
<td>1 (99.9%)</td>
</tr>
<tr>
<td>ft</td>
<td>3 (&lt;1%)</td>
<td>37 (88.2%)</td>
<td>6 (88.2%)</td>
</tr>
<tr>
<td>is</td>
<td>8 (29.4%)</td>
<td>9 (28.5%)</td>
<td>1 (27.3%)</td>
</tr>
<tr>
<td>lu</td>
<td>88 (65.9%)</td>
<td>54 (99.7%)</td>
<td>29 (81.5%)</td>
</tr>
<tr>
<td>mg</td>
<td>9 (4.70%)</td>
<td>48 (77.7%)</td>
<td>12 (77.7%)</td>
</tr>
<tr>
<td>sp</td>
<td>178 (88.0%)</td>
<td>287 (99.6%)</td>
<td>70 (61.8%)</td>
</tr>
<tr>
<td>equake</td>
<td>29 (23.8%)</td>
<td>69 (98.1%)</td>
<td>11 (98.0%)</td>
</tr>
<tr>
<td>art</td>
<td>16 (30.0%)</td>
<td>31 (85.6%)</td>
<td>5 (65.0%)</td>
</tr>
<tr>
<td>ammp</td>
<td>43 (&lt;1%)</td>
<td>21 (1.40%)</td>
<td>7 (84.4%)</td>
</tr>
</tbody>
</table>
Parallelism Coverage

- Profile-driven: coverage close to manually parallelized

<table>
<thead>
<tr>
<th>Application</th>
<th>icc</th>
<th>Profile-driven</th>
<th>Manual</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#loops(%cov)</td>
<td>#loops(%cov)</td>
<td>#loops(%cov)</td>
</tr>
<tr>
<td>bt</td>
<td>72 (18.6%)</td>
<td>205 (99.9%)</td>
<td>54 (99.9%)</td>
</tr>
<tr>
<td>cg</td>
<td>16 (1.10%)</td>
<td>28 (93.1%)</td>
<td>22 (93.1%)</td>
</tr>
<tr>
<td>ep</td>
<td>6 (&lt;1%)</td>
<td>8 (99.9%)</td>
<td>1 (99.9%)</td>
</tr>
<tr>
<td>ft</td>
<td>3 (&lt;1%)</td>
<td>37 (88.2%)</td>
<td>6 (88.2%)</td>
</tr>
<tr>
<td>is</td>
<td>8 (29.4%)</td>
<td>9 (28.5%)</td>
<td>1 (27.3%)</td>
</tr>
<tr>
<td>lu</td>
<td>88 (65.9%)</td>
<td>54 (99.7%)</td>
<td>29 (81.5%)</td>
</tr>
<tr>
<td>mg</td>
<td>9 (4.70%)</td>
<td>48 (77.7%)</td>
<td>12 (77.7%)</td>
</tr>
<tr>
<td>sp</td>
<td>178 (88.0%)</td>
<td>287 (99.6%)</td>
<td>70 (61.8%)</td>
</tr>
<tr>
<td>equake</td>
<td>29 (23.8%)</td>
<td>69 (98.1%)</td>
<td>11 (98.0%)</td>
</tr>
<tr>
<td>art</td>
<td>16 (30.0%)</td>
<td>31 (85.6%)</td>
<td>5 (65.0%)</td>
</tr>
<tr>
<td>ammp</td>
<td>43 (&lt;1%)</td>
<td>21 (1.40%)</td>
<td>7 (84.4%)</td>
</tr>
</tbody>
</table>
Parallelism Coverage

- ammp: we fail to parallelize the critical loop

<table>
<thead>
<tr>
<th>Application</th>
<th>icc #loops(%cov)</th>
<th>Profile-driven #loops(%cov)</th>
<th>Manual #loops(%cov)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bt</td>
<td>72 (18.6%)</td>
<td>205 (99.9%)</td>
<td>54 (99.9%)</td>
</tr>
<tr>
<td>cg</td>
<td>16 (1.10%)</td>
<td>28 (93.1%)</td>
<td>22 (93.1%)</td>
</tr>
<tr>
<td>ep</td>
<td>6 (&lt;1%)</td>
<td>8 (99.9%)</td>
<td>1 (99.9%)</td>
</tr>
<tr>
<td>ft</td>
<td>3 (&lt;1%)</td>
<td>37 (88.2%)</td>
<td>6 (88.2%)</td>
</tr>
<tr>
<td>is</td>
<td>8 (29.4%)</td>
<td>9 (28.5%)</td>
<td>1 (27.3%)</td>
</tr>
<tr>
<td>lu</td>
<td>88 (65.9%)</td>
<td>54 (99.7%)</td>
<td>29 (81.5%)</td>
</tr>
<tr>
<td>mg</td>
<td>9 (4.70%)</td>
<td>48 (77.7%)</td>
<td>12 (77.7%)</td>
</tr>
<tr>
<td>sp</td>
<td>178 (88.0%)</td>
<td>287 (99.6%)</td>
<td>70 (61.8%)</td>
</tr>
<tr>
<td>equake</td>
<td>29 (23.8%)</td>
<td>69 (98.1%)</td>
<td>11 (98.0%)</td>
</tr>
<tr>
<td>art</td>
<td>16 (30.0%)</td>
<td>31 (85.6%)</td>
<td>5 (65.0%)</td>
</tr>
<tr>
<td>ammp</td>
<td>43 (&lt;1%)</td>
<td>21 (1.40%)</td>
<td>7 (84.4%)</td>
</tr>
</tbody>
</table>
Safety

- Inherently unsafe, but surprisingly **no FP**
- Even when trained on the **smallest** dataset

<table>
<thead>
<tr>
<th>Application</th>
<th>FP</th>
<th>FN</th>
</tr>
</thead>
<tbody>
<tr>
<td>bt</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>cg</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ep</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ft</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>is</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lu</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mg</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>sp</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>equake</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>art</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ammp</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
- Intel ICC **fails** to deliver any performance gain
- Even **slowdown** for some benchmarks
Profile-driven parallelization achieves 96% of the performance of manually parallelized benchmarks!
In this section we examine the effectiveness of three mapping schemes: manual heuristic with static features and machine learning classifier has decided not to parallelize all data sets and therefore are never profiled. The third one is a profile-driven approach outperforms ICC and the fixed heuristic. On average, our machine learning based approach outperforms ICC and the fixed heuristic for parallelized SPEC OMP codes and, in some cases, is able to outperform them. The static approach performs poorly and is unable to accurately determine whether a loop should be parallelized or not using only static code features and a linear work model is unable to determine whether a loop should be parallelized or not using only static code features. The main reason for this performance loss is that the default scheme appears to be largely due to non-optimal mapping decisions. To address this, we developed a machine learning-based approach that accurately determines whether a loop should be parallelized or not using only static code features and a linear work model. This approach outperforms ICC on average by 84%, rather than 94%, for the NAS benchmarks. Additionally, our machine learning-based approach can outperform the static approach by a marginal improvement over the static mapping scheme while achieving accurate parallelization decisions. In the case of ICC, we show the performance of two different mapping approaches. By default, ICC employs a compile-time profitability approach that decides against a static threshold. While our machine learning approach outperforms ICC, it also requires more runtime checks to be performed.

### Table 4

<table>
<thead>
<tr>
<th>Function</th>
<th>ICC</th>
<th>Manual Parallelization</th>
<th>Profile-driven Parallelization</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPEC OMP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.F</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.G</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.H</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.I</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.J</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.K</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.L</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.M</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.P</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.Q</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.U</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.V</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.W</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FT.Z</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAS PB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPECOMP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Speedup (Intel Xeon)

- SPECOMP 2001 benchmarks include additional sequential optimisations besides parallelisation.
Speedup (Intel Xeon)

- SPECOMP single-threaded has average speedup of 2x over SPECFP due to sequential optimisations
• ammp: critical loop not parallelised by profile-driven technique. Misclassification by ML.

• Manual parallelisation: **1.6x** on 8 cores
- **SPECCOMP.art** \textbf{3.34x} with \textbf{1 thread}
- Profile-driven parallelisation delivers \textbf{4x} speedup without sequential optimisation

![Speedup (Intel Xeon)](image-url)
PART 3: EXTRACTION OF PIPELINE PARALLELISM
Observations

• There is more parallelism available beyond parallel FOR loops
  • Programmers exploit coarse-grained parallelism routinely
  • Auto-parallelising compilers don’t!

• Parallel Design Patterns
  • Static: Pipelines, Task Graphs
  • Dynamic: Task Farms, Divide & Conquer, ...

• Serious Programme Restructuring Required
  • Let’s Do It!
Motivating Example

EEMBC mp3player Algorithmic components

```c
while(end) {
    /* ...input... */
    decode_info(&bs, &fr_ps);
    /* ...input... */
    III_get_side_info(&bs, &III_side_info, &fr_ps);

    for (gr = 0; gr < max_gr; gr++) {
        for (ch = 0; ch < stereo; ch++) {
            III_get_scale_factors(gr, ch, ...);
            III_huffman_decode(gr, ch, ...);
            III_dequantize_sample(gr, ch, ...);
        }
    }
    /* ch */

    III_stereo(gr, ...);

    for (ch = 0; ch < stereo; ch++) {
        III_reorder(ch, gr, ...);
        III_antialias(ch, gr, ...);
    }
    /* sb */

    for (sb = 0; sb < SBLIMIT; sb++) {
        III_hybrid(sb, ch, ...);
    }
    /* ss */

    for (ss = 0; ss < SSLIMIT; ss++) {
        for (sb = 0; sb < SBLIMIT; sb++) {
            if ((ss % 2) && (sb % 2))
                polyPhaseIn[sb] = -hybridOut[sb][ss];
            else
                polyPhaseIn[sb] = hybridOut[sb][ss];
        }
    }
    /* sb */
    clip += SubBandSynthesis(ch, ss, ...);
    /* ss */
    for (ch = 0; ch < stereo; ch++) {
        III_stereo(gr, ...);
    }
    /* ch */
}
/* while */
```

EEMBC mp3player
input, header info 2%
Huffman decoding 5%
dequantize 44%
stereo <1%
reorder <1%
antialias <1%
hybrid 22%
subband synthesis 24%
output 2%
Motivating Example

**EEMBC mp3player**  Algorithmic components

```c
while (end) {
    /* ...input... */
    decode_info(&bs, &fr_ps);
    /* ...input... */
    III_get_side_info(&bs, &III_side_info, &fr_ps);

    for (gr = 0; gr < max_gr; gr++) {
        for (ch = 0; ch < stereo; ch++) {
            III_get_scale_factors(gr, ch, ...);
            III_hufman_decode(gr, ch, ...);
            III_dequantize_sample(gr, ch, ...);
        } /* ch */
        III_stereo(gr, ...);

        for (ch = 0; ch < stereo; ch++) {
            III_reorder(ch, gr, ...);
            III_antialias(ch, gr, ...);

            for (sb = 0; sb < SBLIMIT; sb++) {
                III_hybrid(sb, ch, ...);
            } /* sb */
            Level 2

            for (ss = 0; ss < SSLIMIT; ss++) {
                for (sb = 0; sb < SBLIMIT; sb++) {
                    if ((ss % 2) && (sb % 2))
                        polyPhaseIn[sb] = -hybridOut[sb][ss];
                    else
                        polyPhaseIn[sb] = hybridOut[sb][ss];
                } /* sb */
            } /* ss */
        } /* ch */
    } /* gr */

    out_fifo(*pcm_sample, ...);
} /* while */
```
Motivating Example

EEMBC mp3player

Single-level partitioning

input, decoding \[98\%\]

output \[2\%\]

Single-level pipeline is inefficient!
Motivating Example

EEMBC mp3player

```c
while(!end) {
    /* ...input... */
    decode_info(&bs, &fr_ps);
    /* ...input... */
    III_get_side_info(&bs, &III_side_info, &fr_ps);

    for (gr = 0; gr < max_gr; gr++) {
        for (ch = 0; ch < stereo; ch++) {
            III_get_scale_factors(gr, ch, ...);
            III_huffman_decode(gr, ch, ...);
            III_dequantize_sample(gr, ch, ...);
        } /* ch */
    } /* gr */

    III_stereo(gr, ...);

    for (ch = 0; ch < stereo; ch++) {
        III_reorder(ch, gr, ...);
        III_antialias(ch, gr, ...);
        for (sb = 0; sb < SBLIMIT; sb++) {
            III_hybrid(sb, ch, ...);
        } /* ss */
    } /* ch */

    for (ss = 0; ss < SSLIMIT; ss++) {
        for (sb = 0; sb < SBLIMIT; sb++) {
            if ((ss % 2) && (sb % 2))
                polyPhaseIn[sb] = -hybridOut[sb][ss];
            else
                polyPhaseIn[sb] = hybridOut[sb][ss];
        } /* sb */
    } /* ss */

    out_fifo(*pcm_sample, ...);
} /* while */
```

Multi-level partitioning

- header info, huffman dec. 7%
- dequantize 44%
- stereo, reorder, antialias, hybrid 23%
- subband synthesis 24%
- output 2%

speedup potential 2.27x
Motivating Example

### EEMBC mp3player

```c
while(!end) {
    decode_info(&bs, &fr_ps);
    III_decode_side_info(&bs, &III_side_info, &fr_ps);

    for (gr = 0; gr < max_gr; gr++) {
        for (ch = 0; ch < stereo; ch++) {
            III_get_scale_factors(gr, ch, ...);
            III_huffman_decode(gr, ch, ...);
            III_dequantize_sample(gr, ch, ...);
        } /* ch */
        III_stereo(gr, ...);
    } /* gr */

    for (ch = 0; ch < stereo; ch++) {
        III_reorder(ch, gr, ...);
        III_antialias(ch, gr, ...);
    } /* ch */

    for (sb = 0; sb < SBLIMIT; sb++) {
        III_hybrid(sb, ch, ...);
    } /* sb */

    for (ss = 0; ss < SSLIMIT; ss++) {
        for (sb = 0; sb < SBLIMIT; sb++) {
            if ((ss % 2) && (sb % 2))
                polyPhaseIn[sb] = -hybridOut[sb][ss];
            else
                polyPhaseIn[sb] = hybridOut[sb][ss];
        } /* sb */
        clip += SubBandSynthesis(ch, ss, ...);
    } /* ss */
}
```

### Multi-level + Replication Partitioning

- **header info, huffman dec.**: 7%
- **dequantize**: 22%
- **stereo, reorder, antialias, hybrid**: 23%
- **subband synthesis**: 24%
- **output**: 2%

---

Motivating Example

- **speedup potential**: 4.16x
Motivating Example

**Hierarchical** pipelines increase efficiency

**Replication** of pipeline stages exposes additional parallelism

**Orthogonal** to traditional parallelisation approaches (parallel loops inside pipeline stages)

---

```c
while(end) {
    /* ...input... */
    decode_info(&bs, &fr_ps);
    /* ...input... */

    for (int i = 0; i < num_elements; i++) {
        polyPhaseIn[sb] = hybridOut[sb][ss];
        /* sb */
        clip += SubBandSynthesis(ch, ss, ...);
        /* ss */
        } /* ch */
        /* gr */
    out_fifo(*pcm_sample, ...);
    /* while */
}
```

---

Multi-level + Replication

**Partitioning**

speedup potential **4.16x**
Approach

Sequential code in C
Profile-driven analysis

PDG
Hierarchical partitioning

Multiple CFGs
Privatization
Communication

Parallel code in C
User approval

Native Parallel Binary

Sequential CFG
Sequentialization
Partitioning Strategy

- Pipeline performance determined by the **slowest stage**
- Apply **code transformations** only to the slowest stage to uncover further parallelism
Partitioning Algorithm

- Top-down approach: loops and functions *folded*
- Preprocess PDG of the loop:
  - Form Strongly Connected Components
  - Focus on slowest component:
    - If data-parallel (i) greedily augment it, and (ii) replicate until another component is the slowest
    - If not data-parallel try to reduce the execution time by *unfolding* loop/function nodes in the component
  - Partition pipeline using the load of the slowest component as threshold
Partitioning Operations

- Loop/Function unfolding
  - “Opening up” loop/function for hierarchical partitioning
- Replication
  - Duplication of partitioning unit for parallel execution
- Split function
  - Insert pipeline stage boundary within function body
- Augment block
  - Merge separate blocks into single pipeline stage
Example

- `loadAndRLEsource()` 15%
- `doReversibleTransformation()` 52%
- `moveToFrontCodeAndSend()` 33%
Example

- loadAndRLEsource() 15%
  - doReversibleTransformation() 52%
    - moveToFrontCodeAndSend() 33%

replicate x2

- loadAndRLEsource() 15%
  - doReversibleTransformation() 26%
    - moveToFrontCodeAndSend() 33%

x2
Example
Example

- loadAndRLEsource() 15%
- doReversible Transformation() 52%
- moveToFrontCodeAndSend() 33%

- replicate x2

- loadAndRLEsource() 15%
- doReversible Transformation() 26%
- moveToFrontCodeAndSend() 33%

- split function

- loadAndRLEsource() 15%
- doReversible Transformation() 26%
- generateMTFValues() 17%
- sendMTFValues() 16%

- augment & replicate x5
Example

- loadAndRLEsource() 15%
- doReversibleTransformation() 52%
- moveToFrontCodeAndSend() 33%

replicate x2

- loadAndRLEsource() 15%
- doReversibleTransformation() 26%
- moveToFrontCodeAndSend() 33%

split function

- loadAndRLEsource() 15%
- doReversibleTransformation() 26%
- generateMTFValues() 17%
- sendMTFValues() 16%

augment & replicate x5

- loadAndRLEsource() 15%
- doReversibleTransformation(), generateMTFValues() ~14%
- generate coding tables & selectors 12%
- output 4%
### Experimental Evaluation

#### Application

<table>
<thead>
<tr>
<th>Application</th>
<th>source</th>
<th>lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP3 decode</td>
<td>EEMBC 2.0</td>
<td>20K</td>
</tr>
<tr>
<td>MPEG-2 decode</td>
<td>EEMBC 2.0</td>
<td>23K</td>
</tr>
<tr>
<td>JPEG encode</td>
<td>EEMBC 2.0</td>
<td>22K</td>
</tr>
<tr>
<td>bzip2 compress</td>
<td>SPEC CPU2000</td>
<td>5K</td>
</tr>
</tbody>
</table>

#### Evaluation platform

| Hardware          | Dual Socket, Intel Xeon X5450 @ 3.00GHz  
|                   | 2 Quad-cores, 8 cores in total 
|                   | SSE2, SSE3 and SSE4.1 extensions  
|                   | 6Mb L2-cache shared/2 cores (12Mb/chip)  
|                   | 16Gb DDR2 SDRAM  
| O.S.              | 64-bit Scientific Linux  
|                   | kernel 2.6.9-55 x86_64  
| Compiler          | GNU GCC 4.4.1  
|                   | -O3 -march=core2  

the execution time of the slowest stage.

which is equipped with 8 cores in total. Note that some applications utilise less than 8 cores

This benchmark implements a decoder for the de-facto standard for digital music com-

pression.

dequantize

decode MB

motion compensation

output

input, color convert, downsample

fwd DCT, quantize, encode, output

input, RLE

BW transform, generate MTF, generate tables

output

Table 6.2:

The key challenge in parallelising this application is in exposing su-

efficient work spread

bottleneck stage

replicable stage

List of the benchmarks used for evaluation and their main characteristics. For all

MP3 decoding

MPEG-2 decoding

JPEG encoding

bzip2 compression

MP3

Mpeg-1

JPEG

bzip2 compression

22%

23%

24%

22%

23%

24%

2%
Performance Results

<table>
<thead>
<tr>
<th>Application</th>
<th>replication</th>
<th>multi-loop</th>
<th>func. split</th>
<th># cores</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>MP3 dec.</td>
<td>✓</td>
<td>✓</td>
<td>–</td>
<td>7</td>
<td>3.52x</td>
</tr>
<tr>
<td>MPEG-2 dec.</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>3</td>
<td>2.68x</td>
</tr>
<tr>
<td>JPEG enc.</td>
<td>–</td>
<td>✓</td>
<td>✓</td>
<td>2</td>
<td>1.47x</td>
</tr>
<tr>
<td>bzip2 com.</td>
<td>✓</td>
<td>–</td>
<td>✓</td>
<td>8</td>
<td>4.70x</td>
</tr>
</tbody>
</table>
Further Details

• Sequentialisation of the PDG
• Data privatisation
• Inter-thread communication
• Dynamic memory disambiguation
• Pipeline runtime system
Thanks

• Georgios Tournavitis (now with Intel Research)
• Zheng Wang
• Tobias Edler von Koch
• Igor Böhm
• Damon Fenacci
• Alastair Murray
• Daniel Powell
• Stephen Kyle
• Harry Wagstaff
• Miles Gould
• and my colleagues Mike O’Boyle & Nigel Topham
Summary

• Serious demand for parallelisation tool support
• Static analysis are too conservative
  • Profile driven analyses detect more parallelism, but require additional manual checking
• Mapping of SW parallelism to HW parallelism is non-intuitive and depends on target platform
  • Successful application of machine learning
• More scope for parallelisation beyond FOR loops
  • Start exploiting parallel design patterns
Other Interests

• Everything Parallel
• Code Generation for Embedded Processors
• Fast Instruction Set Simulation
• Parallel JIT Compilation
• Statistical Performance Modelling
• Detection of Parallel Design Patterns
• Mapping for Heterogeneous Multi-Cores
Questions?
BACKUP SLIDES
RESULTS FOR CELL
Performance on Cell

- Overhead is more obvious on small datasets
Performance on Cell

- EP gets significant speedup
  - No synchronization
  - Not memory bound
PIPELINES: CODE GENERATION
Parallel code generation

```c
5%  while((n = read_file(inf, data)) != EOF) {
 20%    for (blk=0; blk<n; blk++) {
      50%      coef[blk] = decode(data, blk);
      5  } 
 20%  out.data = enhance_filter(raw.data);
 5%  write_file(outf, out.data);
}/* while */
```
Parallel code generation

```c
while((n = read_file(inf, data)) != EOF) {
    for (blk=0; blk<n; blk++) {
        coef[blk] = decode(data, blk);
        raw_data[blk] = inv_transform(coef, blk);
    }
    out_data = enhance_filter(raw_data);
} /* while */
```

Sequential CFG
5% while((n = read_file(inf, data)) != EOF) {
20% for (blk=0; blk<n; blk++) {
50%   coef[blk] = decode(data, blk);
   raw_data[blk] = inv_transform(coef, blk);
} inv_transform();
20% out_data = enhance_filter(raw_data);
5% write_file(outf, out_data);
} /* while */
Most applications that exhibit pipeline parallelism will only have a small number of dominating stages, and thus parallelizing these stages will give the best speedup. The configuration of the targeted system is given in Table 1.

The configuration of the targeted system is given in Table 1.
Parallel code generation

```
5% while(n = read_file(inf, data) != EOF) {
20% for (blk=0; blk<n; blk++) {
20% coef[blk] = decode(data, blk);
50% raw_data[blk] = inv_transform(coef, blk);
5% out_data = enhance_filter(raw_data);
8} /* while */
```

Sequential CFG

PDG Partitioning

Sequentialization
PIPELINES: COMMUNICATION
Mixed Speculative Multithreaded Execution Models

Polychronis Xekalakis

THEREFORE

OF DEGREE

Doctor of Philosophy

Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2009

Communication

Stage 1

Stage 2

Stage 3

bb0

n ← read_file()
go to bb1

bb1

pred1 := (n!=-1)
if_pred1 bb7 : bb6

bb6

push(q1, [-, pred1])

bb7

blk := 0
push(q1, [coef, pred1])
go to bb8

bb8

pred2 := (blk<n)
if_pred2 bb9 : bb4

bb9

call decode()
go to bb11

bb11

push(q2, [blk, pred2])
blk := blk + 1
go to bb8

bb10

call inv_transform()
go to bb11

bb11

goto bb8

bb0

[coef, pred1] ← pop(q1)
go to bb1

bb1

if_pred1 bb7 : bb6

bb7

bb6

[blk, pred2] ← pop(q2)
if_pred2 bb10 : bb5

bb10

call enhance_filter()
go to bb3

bb3

call write_file()
go to bb5

bb5

goto bb0

bb5

[raw_data, pred1] ← pop(q3)
go to bb1

bb1

if_pred1 bb2 : end

bb2

call enhance_filter()
go to bb3

bb3

call write_file()
go to bb5

bb5

goto bb0
Communication

Stage 1

bb0
n ← read_file()
goto bb1

bb1
pred1 := (n != -1)
if pred1 bb7 : bb6

bb6
push(q1, [-, pred1])
goto bb0

bb7
blk := 0
push(q1, [coef, pred1])
goto bb8

bb8
pred2 := (blk < n)
if pred2 bb9 : bb4

bb9
call decode()
goto bb5

bb11
push(q2, [blk, pred2])
blk := blk + 1
goto bb8

Stage 2

bb0
[coef, pred1] ← pop(q1)
goto bb1

bb1
if pred1 bb7 : bb6

bb6
push(q3, [-, pred1])
goto bb0

bb7
push(q3, [raw_data, pred1])
goto bb0

bb8
[blk, pred2] ← pop(q2)
if pred2 bb10 : bb5

bb10
call inv_transform()
goto bb11

bb11
goto bb8

Stage 3

bb0
[raw_data, pred1] ← pop(q3)
goto bb1

bb1
if pred1 bb2 : end

bb2
call enhance_filter()
goto bb3

bb3
call write_file()
goto bb5

bb5
goto bb0
Mixed Speculative Multithreaded Execution Models

Polychronis Xekalakis

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2009

Communication

Stage 1

Stage 2

Stage 3

bb0

n ← read_file()
goto bb1

bb6

push(q1, [-, pred1])
goto bb8

bb1

pred1 := (n!=-1)
if pred1 bb7 : bb6

bb7

blk := 0
push(q1, [coef, pred1])
goto bb8

bb8

pred2 := (blk<n)
if pred2 bb9 : bb4

bb9

call decode()
goto bb11

bb4

goto bb5

bb5

push(q2, [-, pred2])
goto bb0

bb11

push(q2, [blk, pred2])
blk := blk + 1
goto bb8

Stage 1 Flowchart

bb0

[coef, pred1] ← pop(q1)
goto bb1

bb1

if pred1 bb7 : bb6

bb7

[blk, pred2] ← pop(q2)
if pred2 bb10 : bb5

bb0

[raw_data, pred1] ← pop(q3)
goto bb1

bb1

if pred1 bb2 : end

bb2

call enhance_filter()
goto bb3

bb3

call write_file()
goto bb5

bb5

goto bb0

Stage 2 Flowchart

bb0

[coef, pred1] ← pop(q1)
goto bb1

bb1

if pred1 bb7 : bb6

bb7

[blk, pred2] ← pop(q2)
if pred2 bb10 : bb5

bb6

push(q3, [-, pred1])
goto bb8

bb7

push(q3, [raw_data, pred1])
goto bb0

bb8

[blk, pred2] ← pop(q2)
if pred2 bb10 : bb5

bb9

call decode()
goto bb11

bb10

call inv_transform()
goto bb11

bb11

goto bb8

bb5

goto bb0

Stage 3 Flowchart
Communication

**Stage 1**

- **bb0**: $n \leftarrow \text{read}_\text{file}()$
  - goto bb1

- **bb1**: pred1 := (n!=-1)
  - if pred1 goto bb6

- **bb6**: push(q1, [-, pred1])

- **bb7**: blk := 0
  - push(q1, [coeff, pred1])
  - goto bb8

- **bb8**: pred2 := (blk<n)
  - if pred2 goto bb9

- **bb9**: call decode()
  - goto bb11

- **bb10**: push(q2, [blk, pred2])
  - blk := blk + 1
  - goto bb8

**Stage 2**

- **bb0**: [coeff, pred1] \leftarrow pop(q1)
  - goto bb1

- **bb1**: if pred1 goto bb6

- **bb6**: push(q1, [coeff, pred1])
  - goto bb11

- **bb7**: [blk, pred2] \leftarrow pop(q2)
  - if pred2 goto bb10

- **bb10**: call inv_transform()
  - goto bb11

**Stage 3**

- **bb0**: [raw_data, pred1] \leftarrow pop(q3)
  - goto bb1

- **bb1**: if pred1 goto bb2

- **bb2**: call enhance_filter()
  - goto bb3

- **bb3**: call write_file()
  - goto bb5

- **bb5**: goto bb0

**Notes:**
- The diagram represents the flow of control and data through different stages of communication processes.
- The stages are labeled as Stage 1, Stage 2, and Stage 3.
- Each block (bb) represents a basic block in the control flow graph.
- The arrows indicate the flow of control.
- Pop and push operations are used to manipulate data.
- The 'call' operations are functional calls to other procedures.
Mixed Speculative Multithreaded Execution Models

Polychronis Xekalakis

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2009

Communication

Stage 1

bb0
n ← read_file()
go to bb1

bb1
pred1 := (n!=-1)
if pred1 bb7 : bb6

bb6
push(q1, [-, pred1])
go to bb8

bb8
pred2 := (blk<n)
if pred2 bb9 : bb4

bb9
call decode()
go to bb11

bb11
push(q2, [-, pred2])
go to bb0

Stage 2

bb0
[coef, pred1] ← pop(q1)
go to bb1

bb1
if pred1 bb7 : bb6

bb6
push(q3, [coefficient, pred1])
go to bb5

bb5
push(q2, [blk, pred2])
blk := blk + 1
go to bb11

bb11
call inv_transform()
go to bb11

Stage 3

bb0
[raw_data, pred1] ← pop(q3)
go to bb1

bb1
call enhance_filter()
go to bb3

bb3
call write_file()
go to bb5

bb5
go to bb0

Stage 1

Stage 2

Stage 3
Communication

Stage 1

Stage 2

Stage 3
Communication

Stage 1

Stage 2

Stage 3

bb0
n ← read_file()
goto bb1

bb1
pred1 := (n!=-1)
if pred1 bb7 : bb6

bb6
push(q1, [-, pred1])
goto bb8

bb7
blk := 0
push(q1, [coef, pred1])
goto bb8

bb8
pred2 := (blk<n)
if pred2 bb9 : bb4

bb4
goto bb5

bb5
push(q2, [-, pred2])
goto bb0

bb9
call decode()
goto bb11

bb11
push(q2, [blk, pred2])
blk := blk + 1
goto bb8

bb10
call inv_transform()
goto bb11

bb11
goto bb8

bb0
[coef, pred1] ← pop(q1)
goto bb1

bb1
if pred1 bb7 : bb6

bb6
push(q3, [-, pred1])
goto bb8

bb7
push(q3, [raw_data, pred1])
goto bb0

bb8
[blk, pred2] ← pop(q2)
if pred2 bb10 : bb5

bb10
call write_file()
goto bb5

bb5
goto bb0

bb1
if pred1 bb2 : end

bb2
call enhance_filter()
goto bb3

bb3
call write_file()
goto bb5

bb5
goto bb0
Mixed Speculative Multithreaded Execution

Polychronis Xekalakis

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2009

Communication

Stage 1

Stage 2

Stage 3

\[ n \leftarrow \text{read}\_file() \]
\[ \text{goto bb1} \]

\[ \text{pred1} := (n\neq -1) \]
\[ \text{if}\_pred1 \text{bb7 : bb6} \]

\[ \text{blk} := 0 \]
\[ \text{push}(q1, [-, \text{pred1}]) \]
\[ \text{goto bb8} \]

\[ \text{pred2} := (\text{blk} < n) \]
\[ \text{if}\_pred2 \text{bb9 : bb4} \]

\[ \text{call decode()} \]
\[ \text{goto bb11} \]

\[ \text{push}(q3, [-, \text{pred1}]) \]
\[ \text{goto bb0} \]

\[ \text{[coef, pred1]} \leftarrow \text{pop}(q1) \]
\[ \text{goto bb1} \]

\[ \text{if}\_pred1 \text{bb7 : bb6} \]

\[ \text{push}(q3, [-, \text{pred1}]) \]
\[ \text{goto bb0} \]

\[ \text{[blk, pred2]} \leftarrow \text{pop}(q2) \]
\[ \text{if}\_pred2 \text{bb10 : bb5} \]

\[ \text{call inv\_transform()} \]
\[ \text{goto bb11} \]

\[ \text{call enhance\_filter()} \]
\[ \text{goto bb3} \]

\[ \text{call write\_file()} \]
\[ \text{goto bb5} \]

\[ \text{goto bb0} \]
Mixed Speculative Multithreaded Execution Models

Polychronis Xekalakis

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2009

Communication

Stage 1

Stage 2

Stage 3
Mixed Speculative Multithreaded Execution

Polychronis Xekalakis

The University of Edinburgh

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

2009

Communication

Stage 1

Stage 2

Stage 3

$\text{n} \leftarrow \text{read\_file()}$

$\text{goto bb1}$

$\text{pred1 := (n!=-1)}$

$\text{if \ pred1 bb7 : bb6}$

$\text{blk := 0}$

$\text{push(}q1, [-, \text{pred1}]\text{)}$

$\text{goto bb8}$

$\text{pred2 := (blk<n)}$

$\text{if \ pred2 bb9 : bb4}$

$\text{call \ decode()}$

$\text{goto bb5}$

$\text{push(q1, [&coef, \text{pred1}])}$

$\text{goto bb8}$

$\text{bb6}$

$\text{push(q2, [-, \text{pred2}])}$

$\text{goto bb0}$

$\text{bb7}$

$\text{bb8}$

$\text{bb9}$

Stage 3

$\text{bb10}$

$\text{call \ inv\_transform()}$

$\text{goto bb11}$

$\text{bb11}$

$\text{goto bb8}$

$\text{bb10}$

$\text{call \ enhance\_filter()}$

$\text{goto bb3}$

$\text{bb3}$

$\text{call \ write\_file()}$

$\text{goto bb5}$

$\text{bb5}$

$\text{goto bb0}$

$\text{bb0}$

$\text{bb1}$

$\text{bb2}$

$\text{bb3}$

$\text{bb4}$

$\text{bb5}$

$\text{bb6}$

$\text{bb7}$

$\text{bb8}$

$\text{bb9}$

$\text{bb10}$

$\text{bb11}$
Communication

**Stage 1**

- `bb0`  
  `n ← read_file()`  
  `goto bb1`

- `bb1`  
  `pred1 := (n!=-1)`  
  `if_pred1 bb7 : bb6`

- `bb6`  
  `push(q1, [-, pred1])`

- `bb7`  
  `blk := 0`  
  `push(q1, [coef, pred1])`  
  `goto bb8`

- `bb8`  
  `pred2 := (blk<n)`  
  `if_pred2 bb9 : bb4`

- `bb9`  
  `call decode()`  
  `goto bb11`

- `bb5`  
  `push(q2, [-, pred2])`  
  `goto bb0`

**Stage 2**

- `bb0`  
  `[coef, pred1] ← pop(q1)`  
  `goto bb1`

- `bb1`  
  `if_pred1 bb7 : bb6`

- `bb6`  
  `push(q3, [-, pred1])`

**Stage 3**

- `bb0`  
  `[raw_data, pred1] ← pop(q3)`  
  `goto bb1`

- `bb1`  
  `if_pred1 bb2 : end`

- `bb2`  
  `call enhance_file()`  
  `goto bb3`

- `bb3`  
  `call write_file()`  
  `goto bb5`

- `bb5`  
  `goto bb0`
mixed speculative multithreaded execution

Polychronis Xekalakis

Thesis

in the

University of Edinburgh Institute of Computing Systems Architecture School of Informatics

2009

Communication

Stage 1

Stage 2

Stage 3

bb0

n ← read_file()
goto bb1

bb1

pred1 := (n!=-1)
if pred1 bb7 : bb6

bb6

push(q1, [-, pred1])

bb7

blk := 0
push(q1, [coeff, pred1])
goto bb8

bb8

pred2 := (blk<n)
if pred2 bb9 : bb4

bb4

goto bb5

bb9

call decode()
goto bb11

bb5

push(q2, [-, pred2])
goto bb0

bb0

[coeff, pred1] ← pop(q1)
goto bb1

bb1

if pred1 bb7 : bb6

bb6

bb7

[coeff, pred1]
push(q3, [raw_data, pred1])
goto bb0

bb0

[raw_data, pred1] ← pop(q3)
goto bb1

bb1

if pred1 bb2 : end

bb2

call enhance_filter()
goto bb3

bb3

call write_file()
goto bb5

bb5

goto bb0

bb7

[[blk, pred2] ← pop(q2)

bb8

if pred2 bb10 : bb5

bb10

call inv_transform()
goto bb11

bb11

goto bb8

bb11

goto bb8

bb5

goto bb0
decode()
goto bb11

bb11

goto bb8

bb5

goto bb0
Communication

Stage 1

bb0
n ← read_file()
goto bb1

bb1
pred1 := (n!=-1)
if pred1 bb7 : bb6

bb6
push(q1, [ -, pred1])

bb7
blk := 0
push(q1, [ & coef, pred1])
goto bb8

bb8
pred2 := (blk<n)
if pred2 bb9 : bb4

bb9
call decode()
goto bb11

bb11
push(q2, [blk, pred2])
blk := blk + 1
goto bb8

Stage 2

bb0
[coef, pred1] ← pop(q1)
goto bb1

bb1
if pred1 bb7 : bb6

bb6
push(q3, [ -, pred1])

bb7
goto bb8

bb8
[blk, pred2] ← pop(q2)
if pred2 bb10 : bb5

bb10
call inv_transform()
goto bb11

bb11
goto bb8

Stage 3

bb0
[raw_data, pred1] ← pop(q3)
goto bb1

bb1
if pred1 bb2 : end

bb2
call enhance_filter()
goto bb3

bb3
call write_file()
goto bb5

bb5
goto bb0
Mixed Speculative Multithreaded Execution Models

Polychronis Xekalakis

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2009

Communication

Stage 1

bb0
n ← read_file()
goto bb1

bb1
pred1 := (n!=-1)
if pred1 bb7 : bb6

bb6
push(q1, [-, pred1])

bb7
blk := 0
push(q1, [coef, pred1])
goto bb8

bb8
pred2 := (blk<n)
if pred2 bb9 : bb4

bb4
goto bb5

bb5
push(q2, [-, pred2])
goto bb0

bb9
call decode()
goto bb11

bb11
push(q2, [blk, pred2])
blk := blk + 1
goto bb8

Stage 2

bb0
[coef, pred1] ← pop(q1)
goto bb1

bb1
if pred1 bb7 : bb6

bb6
push(q3, [-, pred1])
goto bb8

bb7
[blk, pred2] ← pop(q2)
if pred2 bb10 : bb5

bb10
call inv_transform()
goto bb11

bb11
goto bb8

Stage 3

bb0
[raw_data, pred1] ← pop(q3)
goto bb1

bb1
if pred1 bb2 : end

bb2
call enhance_filter()
goto bb3

bb3
call write_file()
goto bb5

bb5
goto bb0
Communication

Stage 1

Stage 2

Stage 3

bb0
n ← read_file()
goto bb1

bb1
pred1 := (n!=-1)
if pred1 bb7 : bb6

bb6
push(q1, [-, pred1])

bb7
blk := 0
push(q1, &[coef, pred1])
goto bb8

bb8
pred2 := (blk<n)
if pred2 bb9 : bb4

bb4
goto bb5

bb5
push(q2, [-, pred2])
goto bb0

bb9
call decode()
goto bb11

bb11
call inv_transform()
goto bb11

bb10
call write_file()
goto bb5

bb3
call enhance_file() goto bb3

bb2
call enhance_filter()
goto bb3

bb1
if pred1 bb2 : end

bb1
[coef, pred1] ← pop(q1)
goto bb1

bb1
if pred1 bb7 : bb6

bb6
[blk, pred2] ← pop(q2)
if pred2 bb10 : bb5

bb5
push(q3, [raw_data, pred1])
goto bb0

bb7
[raw_data, pred1] ← pop(q3)
goto bb1

bb7
goto bb8

bb8
goto bb8

bb10
goto bb11

bb11
goto bb8

bb5
goto bb0

bb0
[raw_data, pred1] ← pop(q3)
goto bb1

bb1
if pred1 bb2 : end
Mixed Speculative Multithreaded Execution

Polychronis Xekalakis

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2009

Communication

Stage 1

Stage 2

Stage 3

Stage 1:

Stage 2:

Stage 3:
Communication

Stage 1

Stage 2

Stage 3

bb0

n ← read_file()
goto bb1

bb1

pred1 := (n!=-1)
if pred1 bb7 : bb6

bb6

push(q1, [-, pred1])

bb7

blk := 0
push(q1, [coef, pred1])
goto bb8

bb8

pred2 := (blk<n)
if pred2 bb9 : bb4

bb9

call decode()
goto bb11

bb10

call inv_transform()
goto bb11

bb11

goto bb8

bb5

push(q2, [blk, pred2])
goto bb0

bb1

if pred1 bb7 : bb6

bb6

push(q3, [-, pred1])

bb7

goto bb8

bb1

call write_file()
goto bb5

bb5

call enhance_filter()
goto bb3

bb3

call write_file()
goto bb5

bb5

goto bb0
Communication

Stage 1

Stage 2

Stage 3

bb0
n ← read_file()
goto bb1

bb1
pred1 := (n!=-1)
if pred1 bb7 : bb6

bb6

Fin

[coef, pred1] ← pop(q1)
goto bb1

bb1
if pred1 bb7 : bb6

bb7
blk := 0
push(q1, [coef, pred1])
goto bb8

bb8
pred2 := (blk<n)
if pred2 bb9 : bb4

bb9
call decode()
goto bb11

bb11
push(q2, [blk, pred2])
blk := blk + 1
goto bb8

bb5
call inv_transform()
goto bb11

bb10
bb5

bb11

bb2
call enhance_filter()
goto bb3

bb3
call write_file()
goto bb5

bb5

bb0
[raw_data, pred1] ← pop(q3)
goto bb1

bb1
if pred1 bb2 : end

bb2

bb3

bb5

bb0

bb1

bb2

bb3

bb5

bb0

bb1

bb2
 Mixed Speculative Multithreaded Execution Models

Polychronis Xekalakis

THESIS

OF

EDINBURGH

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2009

Communication

Stage 1

bb0
n ← read_file()
goto bb1
bb1
pred1 := (n!=-1)
if pred1 bb7 : bb6

bb6
push(q1, [-, pred1])

bb7
blk := 0
push(q1, [&coef, pred1])
goto bb8

bb8
pred2 := (blk<n)
if pred2 bb9 : bb4

bb4
goto bb5
bb5
push(q2, [-, pred2])
goto bb0

bb9
call decode()
goto bb11
bb11
push(q2, [blk, pred2])
blk := blk + 1
goto bb8

bb10
call inv_transform()
goto bb11
bb11
goto bb8

Stage 2

bb0
[coef, pred1] ← pop(q1)
goto bb1
bb1
if pred1 bb7 : bb6

bb6
push(q3, [-, pred1])

bb7
push(q3, [raw_data, pred1])
goto bb0

bb8
[blk, pred2] ← pop(q2)
if pred2 bb10 : bb5
bb10
call enhance_filter()
goto bb3
bb3
call write_file()
goto bb5
bb5
goto bb0

Stage 3

bb1
if pred1 bb2 : end
bb2
call write_file()
goto bb5
bb5
goto bb0

FIN
Communication

Stage 1

bb0
n ← read_file()
goto bb1

bb1
pred1 := (n!=−1)
if pred1 bb7 : bb6

bb6
push(q1, [-, pred1])
goto bb8

bb7
blk := 0
push(q1, [coeff, pred1])
goto bb8

bb8
pred2 := (blk<n)
if pred2 bb9 : bb4

bb9
call decode()
goto bb11

bb10
[raw_data, pred1] ← pop(q3)
goto bb0

bb11
call inv_transform()
goto bb11

FIN

Stage 2

bb0
[coeff, pred1] ← pop(q1)
goto bb1

bb1
if pred1 bb7 : bb6

bb6
push(q3, [-, pred1])
goto bb0

bb7
[blk, pred2] ← pop(q2)
if pred2 bb10 : bb5

bb10
call inv_transform()
goto bb11

bb11
goto bb8

FIN

Stage 3

bb0
[raw_data, pred1] ← pop(q3)
goto bb1

bb1
if pred1 bb5 : end

bb2
call enhance_filter()
goto bb3

bb3
call write_file()
goto bb5

bb5
goto bb0

FIN