Adaptive Execution Techniques for SMT Multiprocessor Systems

Changhee Jung  ETRI
Jaejin Lee  Seoul National University

Outline

- SMT multiprocessor architecture
- Motivation
- Adaptive execution strategies
- Performance evaluation
- Conclusions
SMT Multiprocessor Architecture

- Instructions from different threads execute in parallel
- Identical instruction streams from multiple threads may degrade performance
  - Contend for the same functional unit
- Commercially available: Intel Xeon SMP with hyper-threading

Our Goal & Approach

- Improve the performance of applications that contain conventional loop-level parallelism
- Avoid executing some parallel loops in parallel
- Dynamically change the number of threads to run the parallel loops
Serialize the parallel loop that contains small work.

We have to determine execution mode (# of threads) seriously.
Execution Environment

4P&L Execution Mode

Adaptive Execution Framework

parallel program

preprocessor cost estimation and adaptation code generation

machine specific parameters

adaptive parallel program

compiler for the SMT multiprocessor

executable image
Adaptation Schemes

- Compile-time cost estimation (static cost estimation model)
- Run-time cost estimation (graduated instructions)
- 4P4L loop selection (cache misses per instructions)
- 4P8L loop selection (cycles per instruction)
- 4P4L-4P1L adaptive execution (graduated instructions)
- 4P4L-4P8L adaptive execution
- 4P8L execution time estimation

Static Cost Estimation

- Highly inefficient parallel loops
- A simple model
  - \( W_{body} = \Sigma ( \# of each op \times op\_cost ) \)
  - \( W = n \times W_{body} \)
- If \( W < W_{threshold} \) run it sequentially
- Use heuristics to determine \( W_{threshold} \)
### Determining $W_{\text{threshold}}$

#### Compile-Time Estimated Cost vs. Speedup

![Graph showing compile-time estimated cost vs. speedup]

#### Run-Time Cost Estimation

- The number of instructions executed is proportional to the workload.
- $W =$ the average number of instructions executed in each iteration in the first invocation.
- If $W < W_{\text{threshold}}$, run it sequentially in the following invocations.
- Inefficient loops that cannot be handled by compile-time cost estimation (static cost estimation model) or run-time cost estimation (graduated instructions).

![Diagram showing amount of work]

- 4P1L
- 4P4L
- 4P8L
4P4L Loop Selection

- Loops that are heavily affected by the cache conflicts between two logical processors
- L2/L3 cache misses per instruction (MPI)
- High MPI in 4P4L → high MPI in 4P8L
- If MPI > MPI_{threshold}, run it in 4P4L

Determining MPI_{threshold}

Speedup (4P4L/4P8L) vs. Misses per Instruction
4P8L Loop Selection

- Some loops may perform better in 4P8L mode
- Cycles per instruction (CPI)
  - Partially dependent upon MPI
  - The bigger the CPI, the higher the interference in a single physical processor
  - Small CPI $\rightarrow$ less intra-thread contention
- The decision run is in 4P4L
- If the CPI in 4P4L $< \text{CPI}_{\text{threshold}}$ value, run it in 4P8L

Most Recent with Timing (MRT)

- Uses the recent past behavior of a loop to predict its future behavior

  - If # of graduated instructions $< N_{\text{threshold}}$
    - 4P1L-4P4L MRT
    - Otherwise, 4P4L-4P8L MRT
4P4L-4P1L MRT

- On its second invocation, time it in 4P1L.
- Comparing the two measurements, determine its mode in the next invocation.
- For the remaining invocations, time it again and compare it to its most recent execution time in the other mode.

Amount of work

4P4L-4P8L MRT

- On its second invocation, it is executed in 4P8L and timed again.
- The rest is similar to 4P4L-4P1L MRT.
4P8L Execution Time Estimation

- The larger the number of graduated instructions, the higher the interference between the two logical processors
- L2 and L3 cache misses affect the performance significantly
- Estimate the time in 4P8L with the measurements in 4P4L and determine its mode

Regression Analysis

\[
T_{4P8L} = a \cdot N_{4P8L}^{grad} + b \cdot N_{4P8L}^{L2} + c \cdot N_{4P8L}^{L3} + d
\]

\[
N_{4P8L}^{grad} = a_{grad} \cdot N_{4P4L}^{grad} + b_{grad} \cdot N_{4P4L}^{L2} + c_{grad} \cdot N_{4P4L}^{L3}
\]

\[
N_{4P8L}^{L2} = a_{L2} \cdot N_{4P4L}^{L2} + b_{L2} \cdot N_{4P4L}^{L3}
\]

\[
N_{4P8L}^{L3} = a_{L3} \cdot N_{4P4L}^{L3} + b_{L3}
\]

\[
T_{4P8L} = a' \cdot N_{4P4L}^{grad} + b' \cdot N_{4P4L}^{L2} + c' \cdot N_{4P4L}^{L3} + d'
\]
The Result of Regression Analysis

<table>
<thead>
<tr>
<th>Formulas</th>
<th>$R^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{4P4L} = 0.773 \cdot N_{4P4L}^{expr} - 23.258 \cdot N_{4P4L}^{L2} + 393.006 \cdot N_{4P4L}^{L3} + 2293277$</td>
<td>0.9994</td>
</tr>
<tr>
<td>$N_{4P4L}^{expr} = 1.000 \cdot N_{4P4L}^{expr} - 760536$</td>
<td>0.9999</td>
</tr>
<tr>
<td>$N_{4P4L}^{L2} = 3.206 \cdot N_{4P4L}^{L2} - 150422$</td>
<td>0.9253</td>
</tr>
<tr>
<td>$N_{4P4L}^{L3} = 1.623 \cdot N_{4P4L}^{L3} - 15684$</td>
<td>0.8905</td>
</tr>
<tr>
<td>$T_{4P4L} = 0.773 \cdot N_{4P4L}^{expr} - 74.573 \cdot N_{4P4L}^{L2} + 637.781 \cdot N_{4P4L}^{L3} - 960399$</td>
<td>-</td>
</tr>
</tbody>
</table>

Evaluation Environment

- Implemented in a compiler preprocessor written in Perl
- Applications
  - 10 highly parallel NAS and Spec2K benchmarks (FORTRAN 77)
  - Parallelization information from Polaris
- Compilers
  - Intel Fortran compiler for Windows v8.0
- OS
  - Windows 2003 Server
- Architecture
  - 4 way Intel Xeon MP hyper-threading (1.5 GHz)
- Performance counter library
  - Our own implementation using windows system calls
- Changing the # of threads
  - OpenMP directives
  - Windows system call to map threads to logical processors
Adaptation Schemes

- Compile-time cost estimation (static cost estimation model)
- Run-time cost estimation (graduated instructions)
- 4P4L loop selection (cache misses per instructions)
- 4P8L loop selection (cycles per instruction)
- 4P4L-4P1L MRT (graduated instructions)
- 4P4L-4P8L MRT
- 4P8L execution time estimation

Speedup

- Sequential(4P1L)
- 4P4L
- 4P8L
- Static(4P4L)
- Static(4P8L)
- ADP
- EST
Conclusions

- Presented performance estimation models and techniques for generating adaptive execution code for SMT multiprocessor architectures.
- Our code is about twice and eighteen times faster on average than the original code executed on 4 and 8 logical processors, respectively.
- Adaptive execution techniques are promising and effective at speeding up shared-memory parallel programs for SMT multiprocessors.