POSH:
A Profiler-Enhanced TLS Compiler that Leverages Program Structure

Wei Liu, James Tuck, Luis Ceze, Karin Strauss, and Josep Torrellas
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

Jose Renau
University of California at Santa Cruz
http://masc.soe.ucsc.edu

This work was funded by DARPA-IBM under PERCS.
Introduction

- Chip multiprocessors (CMPs) have arrived
  - Pentium D
  - Opteron dual core
  - Power5
  - Niagara
- How do we speedup hard-to-parallelize single-threaded applications on CMPs?

→ Thread-Level Speculation
Thread-Level Speculation (TLS)

- **TLS Hardware**
  - Tracks data accesses at run-time
  - Detects dependence violations
  - Squashes and restarts tasks

![Sequential vs TLS (no dep violations) vs TLS (dep violations)](image)
Motivation

- Most TLS work consists of proposing architectural variations
  - We need to turn our focus to thread extraction
- Automated TLS compilers are key to TLS acceptance
Main Contributions

- POSH is a fully automated TLS compiler for SPECint class applications
  - Simple
    - Leverages code structure (loops, subroutine continuations) for tasks
  - Effective
    - Average speedup for SPECint applications of 1.28 over sequential on a 4-core CMP
  - Leverages both parallelism and data prefetching
- We evaluated the impact of several key decisions:
  - Task structures
  - Use of a profiler
  - Value prediction
Overview of POSH

Compiler

Profiler

Methodology

Evaluation
Flowchart of the POSH Framework

Compiler Passes in gcc-3.5
All data dependences are carried through memory
- No special hardware for register communication
- Support task `spawn` and `commit` instructions
Outline

- Overview of POSH
- Compiler
- Profiler
- Methodology
- Evaluation
Task Selection

Compiler Passes in gcc-3.5

Task Selection
- Program Structure
- Value Prediction

Spawn Hoisting
- Dependence Restriction
- Placement

Refinement
- Parallelism
- Task Size
- Profiling Feedback

Profiling Feedback
- Generate Task Code

Profiler

The Second Watson Conference on the Interaction between Architecture, Circuits, and Compilers (P=ac^2)
Task Selection

- Break the sequential program into tasks
- Select tasks without regard for nesting
- Select the following structures
  - Subroutine continuations
    - Restrict based on return value and size
  - Loops
    - Consider all loops found using strongly connected components
Spawn Hoisting

- Task Selection
- Program Structure
- Value Prediction
- Dependence Restriction
- Placement

Refinement
- Parallelism
- Task Size
- Profiling Feedback
- Generate Task Code

Profiler

Compiler Passes in gcc-3.5

The Second Watson Conference on the Interaction between Architecture, Circuits, and Compilers (P=ac²)
The spawn instruction is *hoisted* relative to the beginning of a task.

Goal is to provide parallelism and prefetching.

Restrictions on hoisting:
- Spawn after the definition of variables used in the task.
- Spawn must be control safe.
  - We avoid complications of control misspeculation.
Refinement

Compiler Passes in gcc-3.5

Task Selection
Program Structure
Value Prediction

Spawn Hoisting
Dependence Restriction
Placement

Parallelism
Task Size
Profiling Feedback

Generate Task Code
Profiler
Static Task Refinement

- Static refinement eliminates tasks:
  - Determined to be very small
  - Offering little parallelism due to small hoist distance

- Feedback by profiling with small data set size:
  - List of most beneficial tasks
Generate Code For Tasks

Compiler Passes in gcc-3.5
An Example of the Compiler at Work

- First select the task
- Task label, commit
- Identify spawn location
- Pass dependences through memory

```c
int i=0;
...
Loop:
if( i > 99 )
goto Lend;

<LOOP BODY>

i = i + 1;
goto Loop;

Lend:
```
Outline

- Overview of POSH
- Compiler
- Profiler
- Methodology
- Evaluation
The POSH Profiler

Compiler Passes in gcc-3.5

Task Selection
- Program Structure
  - Value Prediction

Spawn Hoisting
- Dependence Restriction
  - Placement

Refinement
- Parallelism
  - Task Size

Profiler
- Profiling Feedback
  - Generate Task Code
Profiler Details

- Runs a TLS binary with all tasks selected by the compiler
  - Uses train input set for SPECint
- Executes the TLS binary *sequentially*
  - No TLS architectural support assumed for generality
- Provides a simple L2 cache model to estimate misses
- Not tied to a fixed number of processors
The Profiler’s Goals

- Preserves parallelism
  - Keeps tasks with good overlap
- Rewards prefetching
  - Keeps tasks that prefetch for themselves

\[ \text{Benefit} = \text{Overlap} + \text{Prefetching} \]
Eliminate tasks using dynamic information
- Small dynamic task size
  - BUT, if a small task spawns a successor, treat it with care
- Small or large spawn hoist distance
  - If too small, then no overlap
  - If too large, then potentially too many data dependences
- Too many squashes per invocation
  - Apply a *prefetching correction* to this rule
  - If *Benefit* is high, keep task regardless of squash rate
Outline

- Overview of POSH
- Compiler
-Profiler
- Methodology
- Evaluation
Simulated Architecture

- 4-core chip multiprocessor
- 4 GHz, 3-issue core
- Detailed cache hierarchy model
  - Per core private 16k L1
  - 1MB Shared L2
  - Speculative data kept in the L1
  - Cache coherence protocol aware of versions
- Main memory latency of 500 cycles
- 12 cycle spawn, 20 cycle squash overhead
- Full program simulation (not just loops)
Applications

- Evaluated SPECint 2000
  - Except gcc, perlbench, eon
- First, apply SGI’s source-to-source optimizer (copt)
- Non-TLS binary compiled with gcc-3.5 using –O2
- TLS binary compiled with POSH
Outline

- Overview of POSH
- Compiler
- Profiler
- Methodology
- Evaluation
TLS Speedup Over Sequential Execution

Achieve an average speedup of 1.28 on SPECints!

Using both forms of tasks is simple and effective.
Profler Effectiveness

Targeting prefetching improves performance
Conclusions

- A TLS compiler utilizing program structure and a profiler delivers good speedups
  - 1.28 average speedup for SPECints
- For better performance, consider both parallelism and prefetching
- We evaluated the impact of several important design decisions
  - Employ both subroutine continuations and loops for best performance
  - Usefulness of the profiler
  - Value Prediction
Thank You
POSH:
A Profiler-Enhanced TLS Compiler that Leverages Program Structure

Wei Liu, James Tuck, Luis Ceze, Karin Strauss, and Josep Torrellas
University of Illinois at Urbana-Champaign
http://iacoma.cs.uiuc.edu

Jose Renau
University of California at Santa Cruz
http://masc.soe.ucsc.edu

This work was funded by DARPA-IBM under PERCS.
Backup Slides
Selection and Hoisting Example

```c
int i = 0;
...

Loop:
    if (i > 99)
        goto Lend;

    <LOOP BODY>

    i = i + 1;

goto Loop;

Lend:
```

```c
int x = 0;
...

Loop:
    if (i > 99)
        goto Lend;

    <LOOP BODY>

    i = i + 1;

goto Loop;

Lend:
```

```c
int x = 0;
...........
...

Loop:
    if (i > 99)
        goto Lend;

    <LOOP BODY>

    i = i + 1;

goto Loop;

Lend:
```

```c
int x = 0;
...........
...

Loop:
    if (i > 99)
        goto Lend;

    <LOOP BODY>

    i = i + 1;

goto Loop;

Lend:
```

```c
int x = 0;
...........
...

Loop:
    if (i > 99)
        goto Lend;

    <LOOP BODY>

    i = i + 1;

goto Loop;

Lend:
```

```c
int x = 0;
...........
...

Loop:
    if (i > 99)
        goto Lend;

    <LOOP BODY>

    i = i + 1;

goto Loop;

Lend:
```

```c
int x = 0;
...........
...

Loop:
    if (i > 99)
        goto Lend;

    <LOOP BODY>

    i = i + 1;

goto Loop;

Lend:
```

```c
int x = 0;
...........
...

Loop:
    if (i > 99)
        goto Lend;

    <LOOP BODY>

    i = i + 1;

goto Loop;

Lend:
```
What is a “Task” in POSH?

- From the perspective of POSH:
  - Begin point
  - Spawn point

- Runtime perspective
  - Dynamic execution between begin points

```c
spawn Task1;
Do_some_work();

Task1:
while( i < 1000 ) {
    spawn Task2;
    ...
    ...
}

Task2:
i++;
}
Do_some_more_work();
```
Estimating Task Performance and Benefit

\[ \text{NewCurrTime} = T_{st} + Ovhd_{squash} \]

\[ \text{Benefit} = \text{Overlap} + \text{Prefetch} - \text{MissOvhd} \]
## Characterization of Profiling

<table>
<thead>
<tr>
<th>App.</th>
<th>#Tasks Before</th>
<th># Elim. For Size</th>
<th># Elim. For Hoist</th>
<th># Elim. For Squashing</th>
<th>#Saved for Pref.</th>
<th># Tasks After</th>
</tr>
</thead>
<tbody>
<tr>
<td>gap</td>
<td>36</td>
<td>0</td>
<td>13</td>
<td>11</td>
<td>7</td>
<td>12</td>
</tr>
<tr>
<td>mcf</td>
<td>17</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>parser</td>
<td>464</td>
<td>47</td>
<td>78</td>
<td>158</td>
<td>6</td>
<td>181</td>
</tr>
<tr>
<td>vpr</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>18</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>139.7</strong></td>
<td><strong>14.8</strong></td>
<td><strong>33.7</strong></td>
<td><strong>55.8</strong></td>
<td><strong>2.1</strong></td>
<td><strong>35.4</strong></td>
</tr>
</tbody>
</table>

The Second Watson Conference on the Interaction between Architecture, Circuits, and Compilers (P=ac²)
Thread-Level Speculation (TLS)

Sequential
- With TLS:
  - Assume no dependences
  - Run in parallel
  - Auto detect violations

```plaintext
for (i = 0; i < n; i++) {
    X[Y[i]] = X[Z[i]]...
}
```

TLS Task A
```plaintext
for (i = 0; i < n/2; i++) {
    X[Y[i]] = X[Z[i]]...
}
```

TLS Task B
```plaintext
for (i = n/2; i < n; i++) {
    X[Y[i]] = X[Z[i]]...
}
```