Compiling for the Multiscalar Architecture

T.N. Vijaykumar

PhD Defense

Computer Sciences Department
University of Wisconsin-Madison
Background

Microprocessors are compute engines for a range of computers

Applications: sequential programs

High performance is important

Exploit Instruction Level Parallelism to achieve performance
Exploiting ILP: State of the art

Search for independent instructions in a window

Point of search

window

Dynamic stream

2-wide processing unit

+ Larger window, wider machine - more parallelism

– Larger window, wider machine - hard to clock fast (21264)

Typical window size: 60-100, machine width: 4-8

© 1997 by T.N. Vijaykumar

Compiling for the Multiscalar Architecture
More Parallelism and Fast Clock?

1. Search at multiple points
2. Split big window and wide machine
3. 1-wide processing units
Multiscalar Architecture: Who does what?

In the implementation that I considered

Deciding the points of search: compiler

Maintaining original program dependencies across little windows

- Register dependencies: compiler
- Control flow: hardware
- Memory dependencies: hardware
Role of the Compiler: Task Selection

Compiler decides the points of search by partitioning programs.
Role of the Compiler: Register Values

Compiler orchestrates register value communication
Role of the Hardware: Control Flow

Program

Hardware predicts one of the exits

Predict Predict

PU 1 PU 2 PU 3

Reg Reg Reg

Task 1 Task 2 Task 3

Misspeculation rollback in hardware

© 1997 by T.N. Vijaykumar

Compiling for the Multiscalar Architecture
Role of the Hardware: Memory Values

Assumes memory independence

Overcome ambiguous dependencies
Contributions

Studied interaction between programs and architecture

Identified fundamental performance issues

Constructed a compiler for the Multiscalar architecture
  - Partition sequential programs into tasks
  - Generate inter-task register communication
  - Schedule register communication
  - Loop restructuring, dead register optimizations
  - Specify program information (register and control)

Evaluated compiler techniques using a simulator for Multiscalar
Overview of the Compiler

Traditional (Gcc) vs. Multiscalar

- Parsing
- Jump Optimization
- Common Sub-Expression
- Loop Optimization

- Register Allocation
- Code Generation

- Loop Restructuring
- Task Selection
- Register Communication
- Scheduling
- Task Annotation

© 1997 by T.N. Vijaykumar
Roadmap

Introduction

Task Selection

Inter-task register communication scheduling

Summary
Multiscalar Tasks

Connected, single entry subgraph of the CFG
Corresponds to a contiguous fragment of the dynamic stream
Basic block, multiple basic blocks, loops, function invocations
Arbitrary control and data dependences
Wide spectrum of choices
Task Selection: Factors

How do tasks impact performance?

Fundamental factors interact with task selection

- Inter-task data dependences
- Inter-task control flow
- Task overheads
- Load imbalance
Inter-task Data Dependence

Code with Data Dependence

Execution 1

Code with Data Dependence

Execution 2

Execution 3

No cycles are wasted

© 1997 by T.N. Vijaykumar

Compiling for the Multiscalar Architecture
Inter-task Control Flow

Execution 1

Execution 2
Load Imbalance

Time

Task 1

Task 2

Stall to commit
Task Size

Small

- Task start/end overheads
- Low degree of parallelism

Large

- Lost opportunity to exploit the parallelism within
- More likely to cause memory misspeculations
- Speculative buffers may overflow

Variation in size

- Load imbalance
Task Selection

Involves many complex factors

NP-Complete [Sarkar for functional programs on Multiprocessors]

Viewed as partitioning the Control Flow Graph (CFG) of program

\[
i = a + b \\
\text{if } (i == n) \text{ then } a = x + y \text{ else } b = x + y \\
c = a + i
\]
Heuristics: Control Flow

Include multiple basic blocks

Take advantage of reconvergent control flow paths

Control number of successors

Graph traversal of the CFG with greedy selection
Heuristics: Control Flow
Heuristics: Task Size

Terminate at loop back edges, function invocations

Suppress short function invocations

Unroll short loops
Heuristics: Data Dependence

Try to “include” data dependencies within tasks

But including one may exclude another

Prioritize with frequency

Schedule those that are “cut”

Dataflow analysis to identify the basic blocks to be included
Heuristics: Data Dependence
Heuristics: Data Dependence
Task Selection Heuristics (OoO PUs)

Effective Instructions per cycle (IPC)

- go
- gcc
- compress
- li
- ijpeg
- perl
- vortex

© 1997 by T.N. Vijaykumar

Compiling for the Multiscalar Architecture
Task Selection Heuristics (OoO PUs)

Effective instructions per cycle (IPC)

- tomcatv
- swim
- su2cor
- hydro2d
- mgrid
- applu
- tub3d
- apsi
- fpppp
- wave5

© 1997 by T.N. Vijaykumar

Compiling for the Multiscalar Architecture
Task Selection Heuristics (in-order PUs)

Effective Instructions per cycle (IPC)

control flow
data dependence

0.0
1.0
2.0
3.0
4.0

go gcc li jpeg perl vortex

© 1997 by T.N. Vijaykumar
Compiling for the Multiscalar Architecture
Task Selection Heuristics (in-order PUs)

![Bar chart showing effective instructions per cycle (IPC) for various applications. Each bar is divided into two segments: one for control flow and one for data dependence. The applications include tomcatv, swim, su2cor, hydro2d, mgrid, applu, tub3d, apsi, fpppp, and wave5. The chart illustrates the performance of these applications in terms of effective instructions per cycle, with a focus on control flow and data dependence.]
Roadmap

Introduction

Task Selection

Inter-task register communication scheduling

Summary
Register Communication: Basics

How do partitions honor original register dependencies?

Distributed register files but single register space

All registers are sent from one PU to next

1. Modified register sent after last modification
2. Unmodified registers sent as they arrive

How does the hardware know? Compiler tells it
Register Communication: Annotation

Which registers are modified?

• **Create Mask**: Registers that may be modified

When can they be sent?

• **Forward Bits**: Send register values

Compiler uses dataflow analysis

Register communication is frequent

• Compiler places sends as early as possible
Register Communication: Example

F: Forward Bit on the instruction
Create Mask: r1, r2, r3

© 1997 by T.N. Vijaykumar
Register Communication: Some Details

\[ \text{NODEF}(i, r) = \prod_{j \in \text{children}(i)} \{ \text{NODEF}(j, r) \cap \neg \text{BBDEF}(j, r) \} \]

\[ \text{SEND}(k, r) = \left\{ \sum_{l \in \text{parents}(k)} \neg \text{NODEF}(l, r) \right\} \cap \text{NODEF}(k, r) \]

Initial Values: \( \text{NODEF}(i, r) = \text{TRUE} \)
Register Communication Scheduling

Time   Task1   Task2   Task3
   Fw r1  Rd r1  Rd r2
   Fw r2
Stall

Producer Task
   Fw r1
   move up
   zero stalls

Consumer Task
   Rd r1
   move down
Register Communication Scheduling

Identify the instructions to be moved
  • Producers up, consumers down

Determine how much to be moved
  • Cost model to estimate position

Perform code motion
  • Move code across basic blocks using dataflow analyses
Cost Model

Estimate position by counting #instructions

Dynamic profiling used to obtain frequencies

If loops are included in tasks, calculate position hierarchically

For function invocations use dynamic count
Cost Model

Task 1

\[ \text{Cost: } 5 + 6 + 0.8 \times 5 \]

Task 2

\[ \text{Cost: } 1 \]
Code Motion

\[ r := \text{r dead} \]
Scheduling Example

Task 1

\[ \begin{align*} 
  a &:= x := c + d \\
  r &:= a + b \\
  r &\text{ dead} \\
\end{align*} \]

Task 2

\[ c := r + s \]

\[ x := c + d \]
Scheduling Example

Task 1

a :=
  r := a + b

Task 2

\( c := r + s \)
\( x := c + d \)

\( c := r + s \)
\( x := c + d \)
Loop Restructuring

First iteration without induction increments

Loop Body

i++ branch on i

i++ branch on i

Loop Body

jump back

© 1997 by T.N. Vijaykumar
Compiling for the Multiscalar Architecture
Register Communication Scheduling (OoO)

![Bar chart showing effective instructions per cycle (IPC) for various benchmarks with and without scheduling.](chart.png)
Register Communication Scheduling (i-o)

![Graph showing Effective Instructions per cycle (IPC) for various programs with and without scheduling.](image-url)
Register Communication Scheduling (OoO)
Register Communication Scheduling (i-o)
## Task Size (SPEC95 INT)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Basic Block total</th>
<th>Control Flow control</th>
<th>Data Dependence control</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>099.go</td>
<td>6.4</td>
<td>2.53</td>
<td>2.04</td>
<td>12.7</td>
</tr>
<tr>
<td>124.m88ksim</td>
<td>4.3</td>
<td>2.97</td>
<td>2.42</td>
<td>10.3</td>
</tr>
<tr>
<td>126.gcc</td>
<td>5.8</td>
<td>2.52</td>
<td>2.32</td>
<td>11.6</td>
</tr>
<tr>
<td>129.compress*</td>
<td>5.7</td>
<td>1.78</td>
<td>2.77</td>
<td>15.0</td>
</tr>
<tr>
<td>130.li</td>
<td>3.9</td>
<td>1.89</td>
<td>1.64</td>
<td>7.1</td>
</tr>
<tr>
<td>132.ijpeg</td>
<td>10.6</td>
<td>2.42</td>
<td>2.43</td>
<td>23.8</td>
</tr>
<tr>
<td>134.perl</td>
<td>6.5</td>
<td>2.26</td>
<td>2.20</td>
<td>10.6</td>
</tr>
<tr>
<td>147.vortex</td>
<td>6.9</td>
<td>2.41</td>
<td>2.19</td>
<td>14.0</td>
</tr>
</tbody>
</table>
## Task Size (SPEC95 FP)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Basic Block total</th>
<th>Control Flow control</th>
<th>Data Dependence control</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>101.tomcatv</td>
<td>44.1</td>
<td>4.96</td>
<td>3.24</td>
<td>84.8</td>
</tr>
<tr>
<td>102.swim</td>
<td>42.0</td>
<td>4.13</td>
<td>4.13</td>
<td>87.7</td>
</tr>
<tr>
<td>103.su2cor</td>
<td>49.8</td>
<td>8.03</td>
<td>8.03</td>
<td>107.8</td>
</tr>
<tr>
<td>104.hydro2d</td>
<td>11.9</td>
<td>6.0</td>
<td>5.24</td>
<td>39.5</td>
</tr>
<tr>
<td>107.mgrid</td>
<td>51.4</td>
<td>2.0</td>
<td>2.01</td>
<td>107.4</td>
</tr>
<tr>
<td>110.applu</td>
<td>21.7</td>
<td>1.71</td>
<td>1.70</td>
<td>38.5</td>
</tr>
<tr>
<td>125.turb3d</td>
<td>21.2</td>
<td>2.46</td>
<td>2.44</td>
<td>40.8</td>
</tr>
<tr>
<td>141.apsi</td>
<td>24.8</td>
<td>2.78</td>
<td>2.63</td>
<td>46.8</td>
</tr>
<tr>
<td>145.fpppp*</td>
<td>957.8</td>
<td>1.45</td>
<td>2.50</td>
<td>66.5</td>
</tr>
<tr>
<td>145.wave5</td>
<td>24.4</td>
<td>4.20</td>
<td>4.08</td>
<td>56.1</td>
</tr>
</tbody>
</table>
## Control Flow Misspeculation Rate (INT)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Basic Block</th>
<th>Control Flow</th>
<th>Data Dependence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Task</td>
<td>Branch</td>
</tr>
<tr>
<td>099.go</td>
<td>14.4%</td>
<td>14.7%</td>
<td>5.8%</td>
</tr>
<tr>
<td>124.m88ksim</td>
<td>3.1%</td>
<td>4.0%</td>
<td>1.4%</td>
</tr>
<tr>
<td>126.gcc</td>
<td>4.4%</td>
<td>5.8%</td>
<td>2.3%</td>
</tr>
<tr>
<td>129.compress</td>
<td>5.0%</td>
<td>5.7%*</td>
<td>3.2%*</td>
</tr>
<tr>
<td>130.li</td>
<td>3.3%</td>
<td>4.0%</td>
<td>2.1%</td>
</tr>
<tr>
<td>132.ijpeg</td>
<td>6.0%</td>
<td>3.7%</td>
<td>1.5%</td>
</tr>
<tr>
<td>134.perl</td>
<td>1.6%</td>
<td>3.9%</td>
<td>1.7%</td>
</tr>
<tr>
<td>147.vortex</td>
<td>0.8%</td>
<td>0.7%</td>
<td>0.3%</td>
</tr>
</tbody>
</table>
## Control Flow Misspeculation Rate (FP)

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Basic Block</th>
<th>Control Flow</th>
<th>Data Dependence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Task</td>
<td>Branch</td>
</tr>
<tr>
<td>101.tomcatv</td>
<td>1.6%</td>
<td>0.4%</td>
<td>0.1%</td>
</tr>
<tr>
<td>102.swim</td>
<td>0.1%</td>
<td>0.2%</td>
<td>0.0%</td>
</tr>
<tr>
<td>103.su2cor</td>
<td>3.4%</td>
<td>0.5%</td>
<td>0.1%</td>
</tr>
<tr>
<td>104.hydro2d</td>
<td>0.1%</td>
<td>0.3%</td>
<td>0.1%</td>
</tr>
<tr>
<td>107.mgrid</td>
<td>1.1%</td>
<td>2.2%</td>
<td>1.1%</td>
</tr>
<tr>
<td>110.applu</td>
<td>3.9%</td>
<td>3.9%</td>
<td>2.3%</td>
</tr>
<tr>
<td>125.turb3d</td>
<td>3.4%</td>
<td>5.8%</td>
<td>2.4%</td>
</tr>
<tr>
<td>141.apsi</td>
<td>2.9%</td>
<td>4.3%</td>
<td>1.5%</td>
</tr>
<tr>
<td>145.fpppp</td>
<td>5.6%</td>
<td>1.8%</td>
<td>1.2%</td>
</tr>
<tr>
<td>145.wave5</td>
<td>0.8%</td>
<td>0.8%</td>
<td>0.2%</td>
</tr>
</tbody>
</table>
Roadmap

Introduction

Task Selection

Inter-task register communication scheduling

Summary
Summary

Splits big window and wide machine across several little PUs

Important performance issues:
  • Inter-task control flow
  • Inter-task data dependences (register)
  • Task overheads
  • Load imbalance

Task selection is affects many performance issues
Summary

I constructed a compiler to study program architecture interaction
• Partition a sequential program into tasks
• Generate inter-task register communication
• Schedule inter-task register communication
• Loop restructuring and dead register optimizations
• Provide program information

Task selection heuristics better utilize hardware

Scheduling improves performance modestly to significantly

Both are more important for larger number of PUs
## Comparison with Multiprocessors

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Multiprocessor</th>
<th>Multiscalar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speculative task initiation</td>
<td>No/Difficult</td>
<td>Yes</td>
</tr>
<tr>
<td>Multiple flows of control</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Task determination</td>
<td>Static</td>
<td>Static (possibly dynamic)</td>
</tr>
<tr>
<td>Software guarantee of inter-task control independence</td>
<td>Required</td>
<td>Not required</td>
</tr>
<tr>
<td>Software knowledge of inter-task data dependences</td>
<td>Required</td>
<td>Not required</td>
</tr>
<tr>
<td>Inter-task sync.</td>
<td>Explicit</td>
<td>Implicit/Explicit</td>
</tr>
<tr>
<td>Inter-task communication</td>
<td>Through memory</td>
<td>Through registers and memory</td>
</tr>
<tr>
<td>Register space</td>
<td>Distinct for PEs</td>
<td>Common for PEs</td>
</tr>
<tr>
<td>Memory space</td>
<td>Common</td>
<td>Common for PEs</td>
</tr>
</tbody>
</table>
Superscalar vs. Multiscalar (8-way vs 4x2)

EQUAL CLOCK CYCLE

![Graph showing speedup comparison between non-muxed and 4-muxed for various applications: go, gcc, li, jpeg, perl, m88ksim, compress, vortex. The x-axis represents different applications, and the y-axis represents speedup. The graph shows that 4-muxed generally has higher speedup than non-muxed.]
Superscalar vs. Multiscalar (8-way vs 4x2)

EQUAL CLOCK CYCLE

<table>
<thead>
<tr>
<th>Application</th>
<th>non-muxed</th>
<th>4-muxed</th>
</tr>
</thead>
<tbody>
<tr>
<td>tomcatv</td>
<td></td>
<td></td>
</tr>
<tr>
<td>swim</td>
<td></td>
<td></td>
</tr>
<tr>
<td>su2cor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mgrid</td>
<td></td>
<td></td>
</tr>
<tr>
<td>applu</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tub3d</td>
<td></td>
<td></td>
</tr>
<tr>
<td>apsi</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fpppp</td>
<td></td>
<td></td>
</tr>
<tr>
<td>wave5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Why compiler

Interaction between programs and architecture
Response to program transformations
Flexibility and cost of analyses
Within the limitations of a real compiler implementation