Compiler Optimizations For Highly Constrained Multithreaded Multicore Processors

Xiaotong Zhuang

IBM T.J. Watson Research Center
Agenda

- Overview of My Research
- Processor Model
- Dual-bank Register Allocation
- Inter-thread Register Sharing
- Thread Management
- Other Work
- Future Plan
Overview of Research

Goal:
1. Improve performance
2. Satisfy constraints

Constraints:
architectural design, power, security, reliability

Optimizing compiler
(with architecture co-design)
Topics

Compiler Optimizations
PLDI-06, PLDI-04, PACT-03, PACT-02, LCTES-06, LCTES-03, ACM TECSx2, ICDCS-03

Compiler Optimizations + Architectural Support
PLDI-05, ACM TOPLAS, LCTES-04, ACM TECS, LCTES-04, IPDPS-06

Compiler Optimizations for Security, Secure Architecture
ASPLOS-04, MICRO-06, CASES-04 Best Paper, CGO-06, CGO-05, CASES-05

Others
INFOCOM-06, IPDPS-02, IEEE TPDS, IEEE TOC, ICPP-03
Motivation

- Domain specific multicore processors
  - For special applications
  - Specialized, simplified hardware
  - Complexity pushed to the compiler
  - Thread level parallelism

Examples
- CELL—1 PPE+8 SPUs
- Intel’s 80 core teraflop processor
- Cradle CT3616—16 DSPs+8 GPPs
- ClearSpeed CSX600—96 cores
- Intel IXP
Agenda

- Overview of My Research
- Processor Model
- Dual-bank Register Allocation
- Inter-thread Register Sharing
- Thread Management
- Other Work
- Future Plan
The IXP Processor Model

- SRAM
- SDRAM

- ARM core
- core
- core
- core
- core
Packet Processing Core

- No OS, hardware manages threads
- ALU instructions can finish in 1 cycle
- No cache, long memory latency (30~400 cycles)
- Two banks of registers
- Fast context switch
Compiler Challenges

- Code must be highly efficient (1Gb/s => 400 cycle/packet)

- Architectural constraint—register usage

- Resource constraint—not enough registers
  - Large register file is slow and expensive
  - Memory latency is long
  - Functions are often inlined
  - Threads simultaneously active cannot shared registers

- Service constraint—no OS available
Agenda

- Overview of My Research
- Processor Model
- Dual-bank Register Allocation
- Inter-thread Register Sharing
- Thread Management
- Other Work
- Future Plan
Register Allocation Preliminaries

- **GOAL:** put variables to registers for faster access

- Several variables could be put in the same register if they are active in different places

- Some variables might be put in memory (SPILL) when registers are used up

- **OUTPUT:** for each variable, which register or memory location it should be allocated to
Dual-bank Register Constraint

- Dual-bank Constraint
  - Only for ALU instructions
  - Two source operands must come from different banks
  - Fetching operands in parallel allows 1 cycle latency for all ALU instructions

\[ c = a + b \]

- OR

\[ a \rightarrow \text{bank A}, \quad b \rightarrow \text{bank B} \]

\[ a \rightarrow \text{bank B}, \quad b \rightarrow \text{bank A} \]
Two Issues with Dual-bank Register Assignment

Example 1

\[ a = a + b \]
\[ c = a + c \]
\[ d = b + c \]

\[ a \rightarrow \text{Bank A} \]
\[ b \rightarrow \text{Bank B} \]
\[ c \rightarrow \text{?} \]

Example 2

\[ b = a + b \]
\[ c = a + c \]
\[ d = a + d \]

\[ a \rightarrow \text{Bank A} \]
\[ b \rightarrow \text{Bank B} \]
\[ c \rightarrow \text{Bank B} \]
\[ d \rightarrow \text{Bank B} \]

Intel’s assembly tool leaves these problems to the user!!
Register Conflict Graph (RCG)

- Each variable is a node on the graph
- If two variables appear as SOURCE OPERANDS in at least one ALU instruction, they are connected with a CONFLICT EDGE

The two end nodes of a conflict edge should be in different banks.
Register Conflict Graph (RCG)—Examples

**Example 1**

- \( a = a + b \)
- \( c = a + c \)
- \( d = b + c \)

**Example 2**

- \( b = a + b \)
- \( c = a + c \)
- \( d = a + d \)
No-Conflict Law:

No-Conflict Law:

RCG is conflictless iff RCG is bipartite iff No odd-length cycles

Example 1
Bank A
Bank B

Example 2
Bank A
Bank B

group A

group B

Conflict

Conflict
Detect Odd Cycles up to Certain Length

RCG

breadth-first search tree

Parallel Edge: edge between two nodes at the same level
A level k parallel edge => there is odd cycle of length up to 2k+1
Each node should be a root once; complexity: O(|N| x (|N| + |E|))
Break Odd Cycles with Variable Splitting

code

...=A op B
...=A op C
...=B op C

RCG

- Inserting MOV can split “A” and break the cycle
- Cost: one MOV instruction
In-place Bank Exchange

- Require extra registers, which may not be available.

- Our approaches:
  - If no register, try to free one through rematerialization
  - Last resort: in-place bank exchange

\[
\begin{align*}
\ldots &= RA1 \text{ op } RA2 \\
RA1 &= RA1 \oplus RB \\
RB &= RA1 \oplus RB \\
RA1 &= RA1 \oplus RB \\
\ldots &= RB \text{ op } RA2 \\
RA1 &= RA1 \oplus RB \\
RB &= RA1 \oplus RB \\
RA1 &= RA1 \oplus RB \\
\ldots &= RB \text{ op } RA2 \\
RA1 &= RA1 \oplus RB \\
RB &= RA1 \oplus RB \\
RA1 &= RA1 \oplus RB \\
\ldots &= RB \text{ op } RA2 \\
RA1 &= RA1 \oplus RB \\
RB &= RA1 \oplus RB \\
RA1 &= RA1 \oplus RB \\
\ldots &= RB \text{ op } RA2 \\
RA1 &= RA1 \oplus RB \\
RB &= RA1 \oplus RB \\
RA1 &= RA1 \oplus RB \\
\ldots &= RB \text{ op } RA2 \\
\end{align*}
\]
Breaking Odd Cycles

- Breaking odd cycles with minimal cost is very expensive (NP-complete)
- ILP solver—long compilation time
- A heuristic solution that gives good results quickly
Odd Cycle Breaking Algorithm

1. **Inner-loop:** break all odd-cycle with length $K$ heuristically
2. **Outer-loop:** break all odd-cycles from short to long

- **K=1,** store all splitting patterns to $Pattern\_set$
- **K=K+2**
- **Build breadth-first search tree, put all odd cycles with length $K$ to $Cycle\_set$**
- **Cycle\_set is empty**
- **Find a pattern which breaks m cycles with cost w and m/w is maximal**
- **K+2<=max \#nodes**
- **Y**
- **N**
- **finish**

**End.**
Bank Imbalance

RCG (bipartite graph)

odd cycle breaking

Make $|\text{group A}| = |\text{group B}|$
Near-Balancing

- **GOAL:** roughly balance the two groups, zero cost!
- The graph is likely to be disconnected after cycle breaking.
- Each connected component must be bipartite!

RCG (bipartite graph)

Connected Component 1

Connected Component 2

group A

group B
Solving the Balancing Problem

Suppose the RCG contains \( m \) connected components

The complexity of a naïve but optimal solution is \( O(2^m) \), since each connected component could be “flipped” or “not flipped”

Solving:
- For small \( m \) => exhaustive search \( O(2^m) \)
- For large \( m \) => an approximate algorithm for “subset sum”

Next, fully balance the two banks with a heuristic algorithm
Application of Algebraic Laws

calculate $a + b + c$

$t = a + b$
..$= t + c$

$t = a + c$
..$= t + b$

$t = b + c$
..$= t + a$
Compilation Flowchart

- IXP Assembly Code
  - Our Register Allocator
    - IXP Assembler and Linker
      - Machine Code
Comparison for Number of Spills (Memory Accesses)

- **70% reduction**
- **Completely avoid spills for 5 benchmarks**
Average speedup: 20% purely through compiler optimizations
Compilation time within a few seconds on a Pentium 4 machine
Contributions

- Tackled several hard problems with good, fast solutions
- Achieved 20% speedup through compiler optimizations
- First compiler solution to overcome the dual bank constraint
- Published in PACT-03. This work was later integrated into Intel’s new compiler
Agenda

- Overview of My Research
- Processor Model
- Dual-bank Register Allocation
- Inter-thread Register Sharing
- Thread Management
- Other Work
- Future Plan
Lightweight Context Switch

- Context switch only happens for long latency instructions, highly frequent – every 20 cycles
- Thread execution is non-preemptive, predictable; threads are simultaneously active

1 cycle context switch: only PC is saved
Register Sharing

with traditional context switch

Register File

Thread 1
Thread 2
Thread 3
Thread 4

Our approach

with lightweight context switch

private

shared

Thread 1
Thread 2
Thread 3
Thread 4
What to Put in Shared Registers?

Variables in shared registers must not be used across context switches. Upon context switch, they should already be dead i.e. unused.

Categorize variables into two types: those used across context switches, and those are not; Allocate them separately.
Non-Switch Region (NSR) -- Commbench

**BB1**
- `sum=0`

**BB2**
- If (len<2) br BB6

**BB3**
- `read tmp1 \[buf\], 1`
- `sum+=tmp1&0xFFFF`
- `buf=buf+2`
- `if!(sum&0x80000000) br BB5`

**BB4**
- `sum=(sum&0xFFFF) + (sum>>16)`

**BB5**
- `len-=2`
- `ctx_switch goto BB2`

**BB6**
- `ctx_switch if!(len) goto BB8`

**BB7**
- `read tmp2 \[buf\], 1`
- `sum+=tmp2&0xFFFF`
- `if!(sum>>16) br BB10`

**BB8**
- `if!(sum>>16) br BB10`
- `sum=(sum&0xFFFF) + (sum>>16)`
- `goto BB8`

**BB9**
- `return ~sum`
Non-Switch Region (NSR)--Commbench

BB1: sum=0

BB2: If (len<2) br BB6

BB3:
read tmp1(buf,1)
sum+=tmp1&0xFFFF
buf=buf+2
if!(sum&0x80000000)
br BB5

BB4:
sum=(sum&0xFFFF) +(sum>>16)
len=2
ctx_switch
goto BB2

BB5: ctx_switch
goto BB8

BB6:
ctx_switch
If!(len) goto BB8

BB7:
read tmp2(buf,1)
sum+=tmp2&0xFFFF
If!(sum>>16) br BB10

BB8:
If!(sum>>16) br BB10

BB9:
sum=(sum&0xFFFF) +(sum>>16)
goto BB8

BB10:
return ~sum

BB1, BB2, BB3, BB4, BB5, BB6, BB7, BB8, BB9, BB10

If (len<2) br BB6

If!(len) goto BB8

If!(sum&0x80000000)
br BB5

If!(sum>>16) br BB10

return ~sum
Non-Switch Region (NSR) -- Commbench

BB1: sum = 0

BB2: If (len<2) br BB6

BB3: read tmp1\rightarrow [buf], 1
    sum += tmp1 & 0xFFFF
    buf = buf + 2
    ifl(sum & 0x80000000)
    br BB5

BB4: sum = (sum & 0xFFFF) + (sum >> 16)

BB5: len = 2
    ctx_switch
    goto BB2

BB6: ctx_switch
    Ifl(len) goto BB8

BB7: read tmp2\downarrow [buf], 1
    sum += tmp2 & 0xFFFF

BB8: Ifl(sum >> 16) br BB10

BB9: sum = (sum & 0xFFFF) + (sum >> 16)
    goto BB8

BB10: return ~sum
Non-Switch Region (NSR)--Commbench

BB1: sum=0

BB2: If (len<2) br BB6

BB3:
- read tmp1 j[buf], 1
- sum+=tmp1&0xFFFF
- buf=buf+2
- if!(sum&0x80000000) br BB5

BB4:
- sum=(sum&0xFFFF) +(sum>>16)

BB5:
- len-=2
- ctx_switch
- goto BB2

BB6:
- ctx_switch
- If!(len) goto BB8

BB7:
- read tmp2 j[buf], 1
- sum+=tmp2&0xFFFF

BB8:
- If!(sum>>16) br BB10
- sum=(sum&0xFFFF) +(sum>>16)
- goto BB8

BB9:
- ctx_switch
- return ~sum
Non-Switch Region (NSR)--Commbench

Each Connected Component Form a NSR

```
read tmp1 \[buf\], 1
sum+=tmp1&0xFFFF
buf=buf+2
if!(sum&0x80000000)
br BB5
```

```
len-=2
ctx_switch
goto BB2
```

```
read tmp2 \[buf\], 1
sum+=tmp2&0xFFFF
```

```
if!(len) goto BB8
```

```
return ~sum
```

BB1

BB2

BB3

BB4

BB5

BB6

BB7

BB8

BB9

BB10

BB1

BB2

BB3

BB4

BB5

BB6

BB7

BB8

BB9

BB10
Variable Classification

BB1: sum=0

BB2: if (len < 2) br BB6

BB3: read tmp1 \[buf\], 1
    sum+=tmp1&0xFFFF
    buf=buf+2
    if!(sum&0x80000000) br BB5

BB4: sum=(sum&0xFFFF) + (sum>>16)

BB5: len-=2
    ctx_switch
goto BB2

BB6: ctx_switch
    If!(len) goto BB8

BB7: read tmp2 \[buf\], 1
    sum+=tmp2&0xFFFF
    If!(sum>>16) br BB10

BB8: return ~sum

BB9: sum=(sum&0xFFFF) + (sum>>16)
goto BB8
Inter-thread Register Management

Let all PR=MaxPR and all SR=MaxSR

(*) holds ?

Y

Finish

N

N

N

Reduce Max(SR1,SR2…) by 1 if possible

Reduce PR1 by 1 if possible

Reduce PR2 by 1 if possible

Intra-thd RA

Intra-thd RA

Intra-thd RA

commit the one that incurs min total cost

Reduce Max(SR1,SR2…) by 1 if possible

Reduce PR1 by 1 if possible

Reduce PR2 by 1 if possible

Intra-thd RA

Intra-thd RA

Intra-thd RA

commit the one that incurs min total cost
Four threads with identical code
24% saving on the number of registers needed
128 physical registers
Speedup up to 29%
Average 20%
Contributions

- Partially sharing registers among threads alleviates registers shortage

- Combined with intra-thread allocation, it gives us around 40% speedup

- Published in PLDI-04, later integrated into Intel’s new compiler

- Our recent work on IPDPS-06 adds hardware modifications to achieve more sharing
Agenda

- Overview of My Research
- Processor Model
- Dual-bank Register Allocation
- Inter-thread Register Sharing
- Thread Management
- Other Work
- Future Plan
Motivation

- CPU cycle wastage (20-30%) due to unnecessary stalls

- Need for better CPU sharing, some threads take more CPU due to less long latency instructions

- Real-time scheduling, packet scheduling
Example — Weighted Round Robin

thread 1

Inst 1.1
Inst 1.2
Inst 1.3
Inst 1.4
Inst 1.5
Inst 1.6
...
weight=2

thread 2

Inst 2.1
Inst 2.2
Inst 2.3
Inst 2.4
Inst 2.5
Inst 2.6
...
weight=3

thread 1

Inst 1.1
Inst 1.2
ctx
Inst 1.3
Inst 1.4
ctx
Inst 1.5
Inst 1.6
ctx
weight=2

thread 2

Inst 2.1
Inst 2.2
ctx
Inst 2.3
Inst 2.4
ctx
Inst 2.5
Inst 2.6
ctx
...
weight=3
# Main Results

<table>
<thead>
<tr>
<th>Category</th>
<th>Constraint</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Scheduling</td>
<td>(Weighted) Round Robin—(W)RR</td>
<td>FCS</td>
</tr>
<tr>
<td></td>
<td>Priority Sharing—PS</td>
<td>FCS</td>
</tr>
<tr>
<td>Real-time Scheduling</td>
<td>Rate Monotonic—RM</td>
<td>FCS</td>
</tr>
<tr>
<td></td>
<td>Earliest Deadline First—EDF</td>
<td>DCS</td>
</tr>
<tr>
<td>Packet Scheduling</td>
<td>Priority Class—PC</td>
<td>FCS</td>
</tr>
<tr>
<td></td>
<td>First Come First Serve—FCFS</td>
<td>DCS</td>
</tr>
<tr>
<td></td>
<td>(Weighted) Fair Queueing—(W)FQ</td>
<td>FCS</td>
</tr>
</tbody>
</table>

- Up to 2% slowdown
- Code growth <5%
- Eliminate unnecessary stalls, 20-30% improvement on CPU utilization
Agenda

- Overview of My Research
- Processor Model
- Dual-bank Register Allocation
- Inter-thread Register Sharing
- Thread Management
- Other Work
- Future Plan
Other Compiler Work

- Parallelize load/store instructions [PACT-02] journal version [ACM TECS]

- Auto addressing mode [LCTES-03]

- Manage hidden registers on ARM [LCTES-04]

- Lower power prefetching [LCTES-04], journal version [ACM TECS]
Other Compiler Work

- Differential encoding and register allocation [PLDI-05], journal version [ACM TOPLAS]

- Compiler scheduling of mobile code in a distributed data intensive environment [ICDCS-03]

- Profile-driven optimizations for server applications [PLDI-06]

- Current project at IBM Research: speculative parallelization for Blue Gene/Q and Power 7
Optimization for Security

- Prevent information leakage through the address bus for secure processors [ASPLOS-04] [CASES-04 Best Paper]
  - Address bus information leakage is a severe problem
  - Propose two solutions to remedy it

- Reduce security overhead, improve security strength through compiler/hardware approaches [CGO-06]
  - Apply to secret sharing [CGO-05]
  - Apply to anomaly detection [MICRO-06] [CASES-05]
Some Other Work

- A highly scalable priority queue [IEEE INFOCOM-06]

- Reduce cache pollution via prefetch filtering [ICPP-03], journal revision [IEEE TOC]

- Low latency broadcasting in massive parallel computers [IPDPS-02], journal version [IEEE TPDS]
Agenda

- Overview of My Research
- Processor Model
- Dual-bank Register Allocation
- Inter-thread Register Sharing
- Thread Management
- Other Work
- Future Plan
Multicore

*The number of cores will double every 18 months, with 256-core systems commonplace by 2011*

—Anant Agarwal, MIT

- Program partitioning, speculative parallelization
- Aggressive speculation with multiple parameterized compilation versions
- On-chip memory organization and data layout optimizations
- Compiler scheduling for power and temperature management
Specialization

- Applications in special domains: multimedia, scientific computing, simulation, physics, chemistry, bio-informatics
  - Specially designed hardware
  - Heterogeneous multicore

- Hybrid optimization according to runtime conditions
  - Compiler generates rough optimization strategies
  - Runtime system fills in the details
Security

- Automatic patch generation for large-scale zero-day worms
- Record forensic data w/ hardware support
- Compiler analysis for worm code and system source code
- Generate the patch automatically

- Compiler/architectural approaches for fast identification of malicious inputs
Questions & Answers

That’s All Folks!