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

Constraints:
architectural design, power, security, reliability

Optimizing compiler
(with architecture co-design)

Goal:
1. Improve performance
2. Satisfy constraints
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

core

core
Packet Processing Core

- register bank A
- register bank B
- CPU
- code store
- MMU

- Thread 1
- Thread 2
- Thread n

- 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 \((1\text{Gb/s} \Rightarrow 400\text{ cycle/packet})\)

- 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 \(\Rightarrow\) 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
  - \( a \leftrightarrow \text{bank A} \), \( b \leftrightarrow \text{bank B} \)
  - \( a \leftrightarrow \text{bank B} \), \( b \leftrightarrow \text{bank A} \)
Two Issues with Dual-bank Register Assignment

Example 1

\[
\begin{align*}
a &= a + b \\
c &= a + c \\
d &= b + c \\
\end{align*}
\]

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

\[\text{bank conflicts!} \]

Example 2

\[
\begin{align*}
b &= a + b \\
c &= a + c \\
d &= a + d \\
\end{align*}
\]

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

\[\text{No conflict but not balanced!} \]

Intel’s assembly tool leaves these problems to the user!!
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

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

Example 2

\[
\begin{align*}
\text{b} &= \text{a} + \text{b} \\
\text{c} &= \text{a} + \text{c} \\
\text{d} &= \text{a} + \text{d}
\end{align*}
\]
No-Conflict Law

No-Conflict Law:

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

Example 1

Bank A

Example 2

Bank A

Bank B

group A

group B

Conflict

Conflict
Detect Odd Cycles up to Certain Length

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| \times (|N|+|E|) )
Break Odd Cycles with Variable Splitting

code

RCG

\[ \ldots = A \text{ op } B \]
\[ \ldots = A \text{ op } C \]
\[ \ldots = B \text{ op } C \]

- 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

\[ \text{...} = \text{RA1 op RA2} \]

\[
\begin{align*}
\text{RA1} &= \text{RA1} \oplus \text{RB} \\
\text{RB} &= \text{RA1} \oplus \text{RB} \\
\text{RA1} &= \text{RA1} \oplus \text{RB} \\
\text{RB} &= \text{RA1}_{\text{orig}} \\
\text{RA1} &= \text{RA1}_{\text{orig}} \oplus \text{RB}_{\text{orig}}; \\
\text{RB} &= \text{RA1}_{\text{orig}}; \\
\text{RA1} &= \text{RA1}_{\text{orig}} \oplus \text{RB}_{\text{orig}}; \\
\text{RB} &= \text{RB}_{\text{orig}}; \\
\text{RA1} &= \text{RA1}_{\text{orig}}; \\
\text{RB} &= \text{RB}_{\text{orig}};
\end{align*}
\]
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. **K = 1**
   - Store all splitting patterns to Pattern_set

2. **K = K + 2**
   - If K + 2 ≤ max #nodes, go back to step 1.
   - If Cycle_set is empty, go to the next step.

3. **Inner-loop**: Break all odd-cycle with length K heuristically.

4. **Outer-loop**: Break all odd-cycles from short to long.

   - Find a pattern which breaks m cycles with cost w and m/w is maximal.

   - If Cycle_set is empty, go back to step 2.

   - Otherwise, go back to step 2.
Bank Imbalance

RCG (bipartite graph)

od 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
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 $m$ $\Rightarrow$ exhaustive search $O(2^m)$

- For large $m$ $\Rightarrow$ an approximate algorithm for “subset sum”

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

**Calculate** \( a + b + c \)

<table>
<thead>
<tr>
<th>( t = a + b )</th>
<th>( t = a + c )</th>
<th>( t = b + c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t )</td>
<td>( t )</td>
<td>( t )</td>
</tr>
<tr>
<td>( a + b + c )</td>
<td>( a + c + b )</td>
<td>( b + a + c )</td>
</tr>
</tbody>
</table>

Diagram:

- \( a \rightarrow b \)
- \( t \rightarrow c \)
- \( a \rightarrow b \)
- \( t \rightarrow c \)
- \( a \rightarrow b \)
- \( t \rightarrow c \)
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)
len-=2
ctx_switch
goto BB2

BB5:

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
Non-Switch Region (NSR) -- Commbench

```
read tmp1 &[buf], 1
sum += tmp1 & 0xFFFF
buf = buf + 2
if!(sum & 0x80000000) br BB5
sum = (sum & 0xFFFF) + (sum >> 16)
len -= 2
ctx_switch
goto BB2

If (len < 2) br BB6
sum = 0

read tmp2 &[buf], 1
if !(sum & 0x80000000) br BB10
sum = (sum & 0xFFFF) + (sum >> 16)
goto BB8
return ~sum
```
Non-Switch Region (NSR)--Commbench

BB1

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

cntx_switch

BB7

read tmp2←[buf], 1

ifl(len) goto BB8

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<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
- goto BB8

BB8:
- ifl(sum>>16)br BB10

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

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

Each Connected Component Form a NSR

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

BB7: If (len) goto BB8

BB8: read tmp2←[buf], 1
    sum += tmp2 & 0xFFFF
    If (sum >> 16) br BB10

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

BB10: return ~sum

NSR1

NSR2

NSR3
Variable Classification

```
if (len < 2) br BB6
 sum = 0
 buf = buf + 2
 if !(sum & 0x80000000)
   br BB5
 sum = (sum & 0xFFFF) + (sum >> 16)
 len = len - 2
 ctx_switch
 goto BB2

read tmp1[buf], 1
 sum += tmp1 & 0xFFFF
 buf = buf + 2
 if !(sum & 0x80000000)
   br BB10
 sum = (sum & 0xFFFF) + (sum >> 16)
 goto BB8

return ~sum
```
Inter-thread Register Management

\[ \sum_i PR_i + \text{Max}(SR_1, SR_2, ..., SR_{N_{thd}}) \leq N_{reg} (*) \]

Let all \( PR = \text{MaxPR} \) and all \( SR = \text{MaxSR} \)

\((*)\) holds?  

Y \rightarrow Finish

N \rightarrow Reduce Max(SR1, SR2, ...) by 1 if possible

N \rightarrow Reduce PR1 by 1 if possible

N \rightarrow Reduce PR2 by 1 if possible

N \rightarrow Intra-thd RA

N \rightarrow Intra-thd RA

N \rightarrow Intra-thd RA

commit the one that incurs min total cost
Register Pressure Reduction

- 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  thread 2  thread 1  thread 2

Inst 1.1   Inst 2.1   Inst 1.1   Inst 2.1
Inst 1.2   Inst 2.2   Inst 1.2   Inst 2.2
Inst 1.3   Inst 2.3   Inst 1.3   Inst 2.3
Inst 1.4   Inst 2.4   Inst 1.3   Inst 2.4
Inst 1.5   Inst 2.5   Inst 1.4   Inst 2.4
Inst 1.6   Inst 2.6   ctx       ctx
...
...
weight=2  weight=3

1.1 1.2  ctx 2.1 2.2 2.3  ctx 1.3 1.4  ctx 2.4 2.5 2.6
## 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!