Beyond ILP

Hemanth M
Bharathan Balaji
Multiscalar Processors

Gurindar S Sohi
Scott E Breach
T N Vijaykumar

Hemanth M & Bharathan Balaji
Control Flow Graph (CFG)

- Each node is a basic block in graph
- CFG divided into a collection of tasks
- Each task consists of sequential instruction stream
- Each task may contain branches, function calls, or loops
How Multiscalar can be useful

Figure 2: An Example Control Flow Graph.

Hemanth M & Bharathan Balaji
How Multiscalar can be useful

Figure 2: An Example Control Flow Graph.
Microarchitecture

Figure 1: A Possible Microarchitecture of a Multiscalar Processor.

Hemanth M & Bharathan Balaji
Microarchitecture

Each task is assigned a processing unit

Figure 1: A Possible Microarchitecture of a Multiscalar Processor.

Hemanth M & Bharathan Balaji
Microarchitecture

A copy of register file is maintained in each unit.

Figure 1: A Possible Microarchitecture of a Multiscalar Processor.

Hemanth M & Bharathan Balaji
Microarchitecture

Assigns the tasks to be executed to the executing unit

Figure 1: A Possible Microarchitecture of a Multiscalar Processor.
Microarchitecture

For indicating the oldest and the latest tasks executing

Figure 1: A Possible Microarchitecture of a Multiscalar Processor.

Hemanth M & Bharathan Balaji
Microarchitecture

A unidirectional ring for forwarding data

Figure 1: A Possible Microarchitecture of a Multiscalar Processor.

Hemanth M & Bharathan Balaji
Interleaved data banks provide data to the processing units.
Figure 1: A Possible Microarchitecture of a Multiscalar Processor.

Microarchitecture

Address Resolution Buffer for memory dependences

Hemanth M & Bharathan Balaji
Microarchitecture Summary

• Each task is assigned a processing unit
• A copy of register file is maintained in each unit
• Sequencer assigns the tasks to be executed to the executing unit
• Head and Tail pointers for tasks
• Interleaved data banks
• A unidirectional ring for forwarding data
• Address Resolution Buffer for memory dependences
Issues

• Partitioning of program into tasks, Demarcation of tasks
  – Done during compile time, assign approximately equal size tasks, Task descriptor
• Maintaining sequential semantics
  – Circular queue, Commit tasks in order
• Data dependencies between tasks
  – Create mask and Accum mask.
• Memory dependencies between tasks
  – Address Resolution Buffer, speculative loads
for (indx = 0; indx < BUFSIZE; indx++) {
    /* get the symbol for which to search */
    symbol = SYMVAL(buffer[indx]);

    /* do a linear search for the symbol in the list */
    for (list = listhd; list; list = LNEXT(list)) {
        /* if symbol already present, process entry */
        if (symbol == LELE(list)) {
            process(list);
            break;
        }
    }

    /* if symbol not found in the list, add to the tail */
    if (!list) {
        addlist(symbol);
    }
}

Figure 3: An Example Code Segment.
<table>
<thead>
<tr>
<th>Tag Spec</th>
<th>Branch, Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag1</td>
<td>OUTER</td>
</tr>
<tr>
<td>Tag2</td>
<td>OUTERFALLOUT</td>
</tr>
<tr>
<td>Create mask</td>
<td>$4, $8, $17, $20, $23</td>
</tr>
</tbody>
</table>

**OUTER:**
- `addu $20, $20, 16`
- `ld $23, SYMVAL-16($20)`
- `move $17, $21`
- `beq $17, $0, SKIPINNER`

**INNER:**
- `ld $8, LELE($17)`
- `bne $8, $23, SKIPCALL`
- `move $4, $17`
- `jal process`
- `jump INNERFALLOUT`

**SKIPCALL:**
- `ld $17, NEXTLIST($17)`
- `bne $17, $0, INNER`

**INNERFALLOUT:**
- `release $8, $17`
- `bne $17, $0, SKIPINNER`
- `move $4, $23`
- `jal addlist`

**SKIPINNER:**
- `release $4`
- `bne $20, $16, OUTER`

**OUTERFALLOUT:**
Data Dependencies

- Create mask: The register values each task may produce.
- Accum mask: Union of all create masks of currently active predecessor tasks.
- The last update of the register in the task should be forwarded. Compiler maintains a forward bit.
- Release instruction added to forward non-updated registers
- A Stop bit is maintained at each exit instruction of the task
Registers which may be needed by another task
Used inside the loop only
Release instruction - used for forwarding the data when no updates are made.
for (indx = 0; indx < BUFSIZE; indx++) {
    /* get the symbol for which to search */
    symbol = SYMVAL(buffer[indx]);

    /* do a linear search for the symbol in the list */
    for (list = listhd; list; list = LNEXT(list)) {
        /* if symbol already present, process entry */
        if (symbol == LELE(list)) {
            process(list);
            break;
        }
    }

    /* if symbol not found in the list, add to the tail */
    if (!list) {
        addlist(symbol);
    }
}

Figure 3: An Example Code Segment.
Other points

• Communication of only **live** registers between tasks
  – Compiler is aware of the live ranges of registers
• Conversion of binaries without compiling
  – Determine the CFG
  – Add task descriptors and tag bits to the existing binary
  – Multiscalar instructions, and relative shift in addresses
Address Resolution Buffer

- Keeps track of all the memory operations in the active tasks.
- Write to memory when a task commits
- Check for memory dependence violation
- Free ARB by
  - Squashing tasks
  - Wait for the task at the head to advance
- Rename memory for parallel function calls
- Can act as bottleneck, and increase latency because of interconnects
Cycles Wasted

• Non-useful computation
  – Incorrect data value
  – Incorrect prediction
• No computation
  – Intra task data dependence
  – Inter task data dependence
• Idle
  – No assigned task
Reclaiming some wasted cycles

• Non-useful Computation Cycles
  – Synchronization of static global variable updates
  – Early validation of predictions

• No Computation Cycles
  – Intra task dependences
  – Inter task dependences (eg. Loop counter)
  – Load Balancing: maintaining granularity & size
    • A function should be distinguished as a suppressed or a full-fledged function

Hemanth M & Bharathan Balaji
Comparison with other paradigms

• Traditional ILP
  – Branch prediction accuracy
  – Monitoring of instructions in the window
  – $N^2$ complexity for dependence cross-checks
  – Identification of memory instructions and address resolution

• Superscalar: Not aware of the CFG

• VLIW: static prediction, large storage name-space, multiported register files, interconnects

• Multiprocessor: tasks have to be independent, no new parallelism discovered

• Multithreaded processor: the threads executing are typically independent of each other

Hemanth M & Bharathan Balaji
Performance Evaluation

- MIPS instruction binaries
- Modified version of GCC 2.5.8
- 5 stage pipeline
- Non-blocking loads and stores
- Single cycle latency for using unidirectional ring
- 32 KB of direct mapped I-cache
- 64 B blocks of $8 \times 2 \times (\text{No. of units})$ KB of direct mapped D-cache
- 256 entries ARB
- PAs predictor for the sequencer

Hemanth M & Bharathan Balaji
# Latencies

<table>
<thead>
<tr>
<th>Integer</th>
<th>Latency</th>
<th>Float</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/Sub</td>
<td>1</td>
<td>SP Add/Sub</td>
<td>2</td>
</tr>
<tr>
<td>Shift/Logic</td>
<td>1</td>
<td>SP Multiply</td>
<td>4</td>
</tr>
<tr>
<td>Multiply</td>
<td>4</td>
<td>SP Divide</td>
<td>12</td>
</tr>
<tr>
<td>Divide</td>
<td>12</td>
<td>DP Add/Sub</td>
<td>2</td>
</tr>
<tr>
<td>Mem Store</td>
<td>1</td>
<td>DP Multiply</td>
<td>5</td>
</tr>
<tr>
<td>Mem Load</td>
<td>2</td>
<td>DP Divide</td>
<td>18</td>
</tr>
<tr>
<td>Branch</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Table 1: Functional Unit Latencies.*
## Increase in the Code Size

<table>
<thead>
<tr>
<th>Program</th>
<th>Instruction Count</th>
<th>Percent Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Scalar</td>
<td>Multiscalar</td>
</tr>
<tr>
<td>Compress</td>
<td>71.04M</td>
<td>81.21M</td>
</tr>
<tr>
<td>Eqntott</td>
<td>1077.50M</td>
<td>1237.73M</td>
</tr>
<tr>
<td>Espresso</td>
<td>526.50M</td>
<td>615.95M</td>
</tr>
<tr>
<td>Gcc</td>
<td>66.48M</td>
<td>75.31M</td>
</tr>
<tr>
<td>Sc</td>
<td>409.06M</td>
<td>460.79M</td>
</tr>
<tr>
<td>Xlisp</td>
<td>46.61M</td>
<td>54.34M</td>
</tr>
<tr>
<td>Tomcatv</td>
<td>582.22M</td>
<td>590.66M</td>
</tr>
<tr>
<td>Cmp</td>
<td>0.98M</td>
<td>1.09M</td>
</tr>
<tr>
<td>Wc</td>
<td>1.22M</td>
<td>1.43M</td>
</tr>
<tr>
<td>Example</td>
<td>1.05M</td>
<td>1.09M</td>
</tr>
</tbody>
</table>

*Table 2: Benchmark Instruction Counts.*

Hemanth M & Bharathan Balaji
# In-Order Results

<table>
<thead>
<tr>
<th>Program</th>
<th>1-Way Issue Units</th>
<th>2-Way Issue Units</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Scalar IPC</td>
<td>Multiscalar</td>
</tr>
<tr>
<td></td>
<td>4-Unit</td>
<td>Speedup</td>
</tr>
<tr>
<td>Compress</td>
<td>0.69</td>
<td>1.17</td>
</tr>
<tr>
<td>Eqntott</td>
<td>0.83</td>
<td>2.05</td>
</tr>
<tr>
<td>Espresso</td>
<td>0.85</td>
<td>1.34</td>
</tr>
<tr>
<td>Gcc</td>
<td>0.81</td>
<td>1.02</td>
</tr>
<tr>
<td>Sc</td>
<td>0.75</td>
<td>1.36</td>
</tr>
<tr>
<td>Xlisp</td>
<td>0.80</td>
<td>0.91</td>
</tr>
<tr>
<td>Tomcatv</td>
<td>0.80</td>
<td>3.00</td>
</tr>
<tr>
<td>Cmp</td>
<td>0.95</td>
<td>3.23</td>
</tr>
<tr>
<td>Wc</td>
<td>0.89</td>
<td>2.37</td>
</tr>
<tr>
<td>Example</td>
<td>0.79</td>
<td>2.79</td>
</tr>
</tbody>
</table>

*Table 3: In-Order Issue Processing Units.*
# Out-of-order Results

<table>
<thead>
<tr>
<th>Program</th>
<th>1-Way Issue Units</th>
<th>2-Way Issue Units</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Scalar IPC</td>
<td>Multiscalar</td>
</tr>
<tr>
<td></td>
<td>Speedup Pred</td>
<td>Speedup Pred</td>
</tr>
<tr>
<td></td>
<td>4-Unit 8-Unit</td>
<td>4-Unit 8-Unit</td>
</tr>
<tr>
<td>Compress</td>
<td>0.72 1.23 86.7%</td>
<td>0.94 1.07 86.7%</td>
</tr>
<tr>
<td>Eqntott</td>
<td>0.84 2.23 94.8%</td>
<td>1.21 1.79 94.8%</td>
</tr>
<tr>
<td>Espresso</td>
<td>0.88 1.47 85.9%</td>
<td>1.31 1.12 85.3%</td>
</tr>
<tr>
<td>Gcc</td>
<td>0.83 1.06 81.1%</td>
<td>1.15 0.91 81.1%</td>
</tr>
<tr>
<td>Sc</td>
<td>0.80 1.42 90.5%</td>
<td>1.10 1.24 90.2%</td>
</tr>
<tr>
<td>Xlisp</td>
<td>0.82 0.95 75.6%</td>
<td>1.12 0.85 74.6%</td>
</tr>
<tr>
<td>Tomcatv</td>
<td>0.96 2.92 99.2%</td>
<td>1.43 2.16 99.2%</td>
</tr>
<tr>
<td>Cmp</td>
<td>0.95 3.24 99.2%</td>
<td>1.68 2.76 99.2%</td>
</tr>
<tr>
<td>Wc</td>
<td>0.89 2.37 99.9%</td>
<td>1.13 2.34 99.9%</td>
</tr>
<tr>
<td>Example</td>
<td>0.86 3.27 99.9%</td>
<td>1.28 2.41 99.9%</td>
</tr>
</tbody>
</table>

Table 4: Out-Of-Order Issue Processing Units.
Results

• Compress: Recurrence and hash table reduce performance (~1.2)
• Eqntott: Loops allow parallel execution(~2.5)
• Espresso: Load balancing & manual granularity(~1.4)
• Gcc: squashes due to incorrect speculation(~0.95)
• Sc: manual function suppression, modified code for load balancing(~1.3)
• Tomcatv, cmp, wc: High parallelism due to loops(>2)
• Example: Again loops, parallelism which cannot be extracted by superscalar processors(~3)
Conclusions

• Multiscalar processor for exploiting more ILP
• Divides the control flow graph into tasks
• Each task is assigned a processing unit, and execution is done speculatively
• Architecture of Multiscalar processor
• Issues related to Multiscalar processor
• Comparison with other paradigms
• More optimizations will improve performance
Simultaneous Multithreading (SMT)

- *Simultaneous Multithreading: Maximizing On-Chip Parallelism*
  Dean M. Tullsen, Susan J. Eggers and Henry M. Levy

- *Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor*
  Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo and Rebecca L. Stamm
From paper to processor


ISCA ‘95

IBM Power5
The Problem with Superscalar

- Horizontal waste = 9 slots
- Vertical waste = 12 slots
Conventional Multithreading

Thread 1
horizontal waste = 9 slots

Thread 2
horizontal waste = 8 slots

No vertical waste
Sources of delay in a wide issue superscalar

- IPC is ~20% of what is possible.
- No one dominant source.
- Attacking each one in turn is painful.
- Vertical waste is only 61%.
Solution: Attack both vertical and horizontal waste

Issue instructions belonging to multiple threads per cycle.
Original Idealized SMT Architecture

- Similar to a Chip Multiprocessor
- Multiple functional units
- Multiple register sets etc.

However, each set need not be dedicated to one thread. Possible models:

1. Fine-Grain Multithreading – Only one thread issues instructions per cycle. Not SMT.
2. Full Simultaneous issue – Any thread can use any number of issue slots – hardware too complex. So,
3. Limit issue slots per thread or
4. Limit functional units connected to each thread context – like CMP
Modified SMT architecture
Modified SMT architecture

1 PC per thread
Modified SMT architecture

Fetch unit can select from any PC

Fetch Unit -> PC

Instruction Cache

8

Decode -> Register Renaming

floating point instruction queue

fp registers -> fp units

fp units

integer instruction queue

integer registers

Data Cache

int/ld-store units
Modified SMT architecture
Modified SMT architecture
Instruction fetch and flow

• Fetch from one PC per cycle in round-robin fashion, ignore stalled threads
• Requires up to $32 \times 8 = 256 + 100$ (for renaming) = 356 registers.
• So spread out register read and write over 2 stages/cycles.
Effects of longer pipeline

- Increases mis-prediction penalty by 1 cycle.
- Extra cycle before write back.
- Physical registers remain bound two cycles longer.
- Some juggling with load instructions to avoid inter-instruction latency.
Cache

• Larger caches required for multithreading to avoid thrashing.
• Authors say not a problem in 5 years. Same goes for register file size.
• Interleaved banks of caches for multiple ports.

<table>
<thead>
<tr>
<th></th>
<th>ICache</th>
<th>DCache</th>
<th>L2 Cache</th>
<th>L3 Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>32KB</td>
<td>32KB</td>
<td>256KB</td>
<td>2MB</td>
</tr>
<tr>
<td>Associativity</td>
<td>DM</td>
<td>DM</td>
<td>4-way</td>
<td>DM</td>
</tr>
<tr>
<td>Line Size</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Banks</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Transfer Time</td>
<td>1 cycle</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Accesses/Cycle</td>
<td>Var (1-4)</td>
<td>4</td>
<td>1</td>
<td>1/4</td>
</tr>
<tr>
<td>Cache fill time</td>
<td>2 cycles</td>
<td>2</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>Latency to next level</td>
<td>6</td>
<td>6</td>
<td>12</td>
<td>62</td>
</tr>
</tbody>
</table>
Performance on SPEC 92

Min of 0.98x – 1 thread
Max of 1.8x IPC

Hemanth M and Bharathan B CSE 240A
Bottlenecks

1. IQ full

- **Fetch Unit**
  - Can fetch 8/cycle,
  - But no room

2. Fetch throughput

- Thread i’s PC

3. Many FUs, but data dependences prevent ||ll issue

- **Fetch Unit**
  - Can fetch 8/cycle,
  - But < 8 ready in this thread

---

Hemanth .M and Bharathan .B CSE 240A
Fixing bottlenecks

2 things we can do with the fetch unit:

1. Fetch from more than one thread in a cycle.
2. Fetch only from threads that will give ‘good’ instructions.

Do these 2 things so that we solve the 3 bottlenecks:

1. Fetch efficiency - Make sure fetch bandwidth utilized.
2. Fetch effectiveness – Make sure ‘good’ instructions fetched keep all FUs busy.
3. Fetch availability – Make sure IQ full-fetching blocked doesn’t happen
Fetch partitioning for fetch efficiency

If 1 thread can’t give 8 instructions at a time, get fewer from more threads per cycle. Possible schemes:

Alg(round robin).num1(No. of threads).num2(No. of instructions/thread)

RR.1.8 – Base, try to get all 8 from a single thread

RR.2.4 or RR.4.2 – Get 4 each from 2 threads or vice versa.

RR.2.8 – Get as many as possible from 1 thread, use thread 2 as backup
Fetch effectiveness for highly parallel instructions

Now we define what good and bad instructions are:

1. Instructions on the wrong path/branch are BAD.
2. Instructions that block others for too long are BAD.

Instruction fetch strategies:

1. **BRCOUNT** – Avoid threads with most unresolved branches.
2. **MISCOUNT** – Avoid threads with most DCache misses pending.
3. **ICOUNT** – Use threads with fewer instructions in pipe.
4. **IQPOSN** – Favor threads with instructions near tail of IQ.

   Use RR to break ties
ICOUNT all the way
Ensuring Fetch Availability

Fetch is blocked when:

1. IQ full or
2. ICache miss

Schemes:

**BIGQ** – Longer IQ, but keep selection pool size same to avoid delay.

**ITAG** – Do I cache tag lookups 1 cycle early, avoid thread next cycle.

**Result** – Using ICOUNT makes this pretty redundant, it is better at removing IQ clog than BIGQ. ITAG gives small improvements.

All put together, we get the promised 2.5x IPC increase over non SMT.
Potential Bottlenecks

1. Issue Bandwidth
   - Issue oldest first (like non SMT)
   - Issue speculative/optimistic last or
   - Branches first.
   
   Results show no big difference. Not a bottleneck.

2. BP & Speculative execution impact single threaded more than SMT.

3. Fetch bandwidth – Can still be improved.

4. IQ Size, Register file size – Effect reduced considerably due to ICOUNT.
Concluding remarks

- Future work involved better compilation techniques for SMT architectures.
- Pentium 4, Atom, Xeon 5500, Power5 all had just 2 threads.
- Nehalem & Itanium 9300 have 8.
- Crypto vulnerability due to shared cache on P4.
- ARM criticism on energy-inefficiency.