Multithreaded Architectures

Uri Weiser and Yoav Etsion
Overview

- Multithreaded Software
- Multithreaded Architecture
- Multithreaded Micro-Architecture
- Conclusions
Multithreading
Basics

- **Process**
  - Each process has its unique address space
  - Can consist of several threads

- **Thread** – each thread has its unique execution context
  - Thread has its own PC (Sequencer) + registers + stack
  - All threads (within a process) share same address space
  - Private heap is optional

- Multithreaded app’s: process broken into threads
  - #1 example: transactions (databases, web servers)
    - Increased concurrency
    - Partial blocking (your transaction blocks, mine doesn’t have to)
    - Centralized (smarter) resource management (by process)
Superscalar Execution
Example – 5 stages, 4 wide machine
Superscalar Execution

Generally increases throughput, but decreases utilization
CMP – Chip Multi-Processor
Multi-threads/Multi-tasks runs on Multi-Processor/core

Time

Low utilization / higher throughput
Multiple Hardware Threads

- A thread can be viewed as a stream of instructions
  - State is represented by PC, Stack pointer, GP Registers

- Equip multithreaded processors with multiple hardware contexts (threads)
  - OS views CPU as multiple logical processors

- Execution of instructions from different threads are interleaved in the pipeline
  - Interleaving policy is critical…
**Blocked Multithreading**

Multi-threading/multi-tasking runs on single core

**Time**

How is this different from OS scheduling?

May increase utilization and throughput, but must switch when current thread goes to low utilization/throughput section (e.g. L2 cache miss)
Fine Grained Multithreading
Multi-threading/multi-tasking runs on single core

Increases utilization/throughput by reducing impact of dependences
Simultaneous Multithreading (SMT)
Multi-threading/multi-tasking runs on single core

Increases utilization/throughput
Blocked Multithreading
(SoE-MT- Switch on Event MT, aka’ – “Poor Man MT”)

- Critical decision: when to switch threads
  - When current thread’s utilization/throughput is about to drop (e.g. L2 cache miss)

- Requirements for throughput:
  - (Thread switch) + (pipe fill time) << blocking latency
    - Would like to get some work done before other thread comes back
  - Fast thread-switch: multiple register banks
  - Fast pipe-fill: short pipe

- Advantage: small changes to existing hardware

- Drawback: high single thread performance requires long thread switch

- Examples
  - Macro-dataflow machine
  - MIT Alewife
  - IBM Northstar
Interleaved (Fine grained) Multithreading

- Critical decision: none?
- Requirements for throughput:
  - Enough threads to eliminate data access and dependencies
    - Increasing number of threads reduces single-thread performance

Superscalar pipeline

Multithreaded pipeline
Interleaved (Fine grained) Multithreading (cont.)

- **Advantages:**
  - (w/ flexible interleave:) Reasonable single thread performance
  - High processor utilization (esp. in case of many thread)

- **Drawback:**
  - Complicated hardware
  - Multiple contexts (states)
  - (w/ inflexible interleave:) limits single thread performance

- **Examples:**
  - HEP Denelcor: 8 threads (latencies were shorter then)
  - TERA: 128 threads
  - MicroUnity - 5 x 1GZ threads = 200 MHz like latency

- **Became attractive for GPUs and network processors**
Simultaneous Multi-threading (SMT)

- Critical decision: fetch-interleaving policy

- Requirements for throughput:
  - Enough threads to utilize resources
    - Fewer than needed to stretch dependences

- Examples:
  - Compaq Alpha EV8 (cancelled)
  - Intel Pentium® 4 Hyper-Threading Technology
SMT Case Study: EV8

- 8-issue OOO processor (wide machine)

- SMT Support
  - Multiple sequencers (PC): 4
  - Alternate fetch from multiple threads
    ✓ Separate register renaming for each thread
  - More (4x) physical registers
  - Thread tags on all shared resources
    ✓ ROB, LSQ, BTB, etc.
    ➔ Allow per thread flush/trap/retirement
  - Process tags on all address space resources: caches, TLB’s, etc.
  - Notice: none of these things are in the “core”
    ✓ Instruction queues, scheduler, wakeup are SMT-blind
Basic EV8

Fetch Decode/Map Queue Reg Read Execute Dcache/Store Buffer Reg Write Retire

PC

Register Map

Icache

Regs

Dcache

Regs

Thread-blind

Computer Architecture 2013 - Multithreading
SMT EV8

Thread-blind

Thread Blind?
Not really
Performance Scalability

Multiprogrammed Workload

SpecInt  SpecFP  Mixed Int/FP

Decomposed SPEC95 Applications

Turb3d  Swm256  Tomcatv

Multithreaded Applications

Barnes  Chess  Sort  TP

Computer Architecture 2013 - Multithreading  19/40
Multithread

I’m afraid of this

![Graph showing SMT Speedup vs Mcycles]

Is CMP a solution?

EV8 simulation R. Espasa
Performance Gain – when?

- When threads do not “step” on each other “predicted state” → $\rightarrow$, Branch Prediction, data prefetch….

- If ample resources are available

- Slow resource, can hamper performance
Scheduling in SMT

- What if one thread gets “stuck”? 

- Round-robin
  - eventually it will fill up the machine (resource contention)

- ICOUNT: thread with fewest instructions in pipe has priority
  - Thread doesn’t get to fetch until it gets “unstuck”

- Other policies have been devised to get best balance, fairness, and overall performance
Scheduling in SMT

- Variation: what if one thread is spinning?
  - Not really stuck, gets to keep fetching
  - Have to stop/slow it artificially (QUIESCE)
    - Sleep until a memory location changes
    - On Pentium 4 – use the Pause instruction
MT Application Data Sharing

- **Shared memory apps**
  - Decoupled caches = Private caches
    - Easier to implement – use existing MP like protocol
  - Shared Caches
    - Faster to communicate among threads
    - No coherence overhead
    - Flexibility in allocating resources

![Data Tag Data Tag Data Tag](image)

<table>
<thead>
<tr>
<th>Way</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Way 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Way 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Way 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Way 7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SMT vs. CMP

- How to use 2X transistors?
  - Better core+SMT or CMP?

- SMT
  - Better single thread performance
  - Better resource utilization

- CMP
  - Much easier to implement
  - Avoid wire delays problems
  - Overall better throughput per area/power
  - More deterministic performance
  - BUT! program must be multithreaded

- Rough numbers
  - SMT:
    ✓ ST Performance/MT performance for 2X transistors = 1.3X/1.5-1.7X
  - CMP:
    ✓ ST Performance /MT performance for 2X transistors = 1X/2X
Intermediate summary

- **Multithreaded Software**

- **Multithreaded Architecture**
  - Advantageous cost/throughput …
  - Blocked MT - long latency tolerance
    - Good single thread performance, good throughput
    - Needs fast thread switch and short pipe
    - OOO execution reduces the advantage!
  - Interleaved MT – ILP increase
    - Bad single thread performance, good throughput
    - Needs many threads; several loaded contexts…
  - Simultaneous MT – ILP increase
    - OK MT performance
    - Good single thread performance
    - Good utilization …
    - Need fewer threads
Example: *Pentium® 4 Hyper-Threading*

- Executes two threads simultaneously
  - Two different applications
  - Two threads of the same application
- CPU maintains architecture state for two processors
  - Two logical processors per physical processor
- Implementation on Intel® Xeon™ Processor
  - Two logical processors for < 5% additional die area
- The processor pretends as if it has 2 cores in an MP shared memory system
uArchitecture impact

- Replicate resources
  - All; per-CPU architectural state
  - Instruction pointers, renaming logic
  - Some smaller resources - e.g. return stack predictor, ITLB, etc.

- Partition resources (share by splitting in half per thread)
  - Several buffers –
    - Re-Order Buffer, Load/Store Buffers, queues, etc.

- Share most resources
  - Out-of-Order execution engine
  - Caches
# Hyper-Threading Performance

- **Performance varies as expected with:**
  - Number of parallel threads
  - Resource utilization
- **Less aggressive MT than EV8 → Less impressive scalability**
  - Machine optimized for single-thread.

<table>
<thead>
<tr>
<th>Application</th>
<th>Performance IXP-MP Hyper-Threading Disabled</th>
<th>Performance IXP-MP Hyper-Threading Enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft* Active Directory</td>
<td>1.00</td>
<td>1.18</td>
</tr>
<tr>
<td>Microsoft* SQL Server</td>
<td>1.22</td>
<td>1.23</td>
</tr>
<tr>
<td>Microsoft* Exchange</td>
<td>1.23</td>
<td>1.30</td>
</tr>
<tr>
<td>Microsoft* IIS</td>
<td>1.30</td>
<td>1.18</td>
</tr>
<tr>
<td>Parallel GCC Compile</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Intel Xeon Processor MP platforms are prototype systems in 2-way configurations**

Applications not tuned or optimized for Hyper-Threading Technology

*All trademarks and brands are the property of their respective owners.*
Multi-Thread Architecture (SoE)

Memory access = P

Execution

Memory access

Execution

Period
Memory latency is shielded by multiple threads execution.
Multi-Thread Architecture (SoE)

Instructions/Memory access=$1/MI$

CPI\textsubscript{i} = CPI ideal of core C

MI = Memory access/Instruction

P = Memory access penalty (cycles)

n = number of threads
Multi-Thread Architecture (SoE)

\[ CPI_i = \frac{P + \left[1/\text{MI}\right] CPI_i}{1/\text{MI}} = CPI_i + \text{MI} \times P \]

- **CPI\(_i\)** = CPI ideal of core C
- **MI** = Memory access/Instruction
- **P** = Memory access penalty (cycles)
- **n** = number of threads

Cycles / instructions
Multi-Thread Architecture (SoE)

Threads architectural states

C

To Memory

Period

\[ IPC_T = \frac{n}{CPI_1} = \frac{n}{[CPI_i + MI \times P]} \]

\( CPI_i = \) CPI ideal of core C

\( MI = \) Memory access/Instruction

\( P = \) Memory access penalty (cycles)

\( n = \) number of threads
Memory latency is shielded by multiple threads execution.

Performance = IPC

Max performance?

Threads architectural states

To Memory
Memory latency is shielded by multiple threads execution.

\[ \text{Performance} = \text{IPC} \]

Max performance = \( \frac{1}{\text{CPI}_i} \)

Multi-Thread Architecture (SoE)

Threads architectural states

\( C \)

To Memory

\( \text{N}_1? \)
The unified machine – the valley

Perf = n/ [ CPI_i + MI * MR(n) * P ]

- Three regions: Cache Region, the valley, MT Region

![Diagram showing performance and threading regions](image-url)
Multi-Thread Architecture (SoE) conclusions

- Applicable when many threads are available
- Require threads state saving
- Can reach “high” performance ($\sim 1/CPI_i$)
- Bandwidth to memory is high
  - $\Rightarrow$ high power
BACKUP SLIDES
The unified machine
Parameter: Cache Size

- Increase Cache size ➔ cache ability to hide memory latency ➔ favors MC

* At this point: unlimited BW to memory
Cache Size Impact

- Increase in cache size $\rightarrow$ cache suffices for more in-flight threads
  - Extends the $\$ region ..AND also $\rightarrow$ Valuable in the MT region
    - Caches reduce off-chip bandwidth $\rightarrow$ delay the BW saturation point

![Graph showing performance for different cache sizes with varying thread counts.](image)
Memory Latency Impact

- Increase in memory latency → Hinders the MT region
- Emphasise the importance of caches

Performance for Different memory Latencies

Unlimited BW to memory

Graph showing performance impact with different memory latencies.