Multithreading: Exploiting Thread-Level Parallelism within a Processor

- Instruction-Level Parallelism (ILP): What we’ve seen so far
- Wrap-up on multiple issue machines
- Beyond ILP
  - Multithreading
Instruction Level Parallelism (ILP): So Far…

Sequential (non pipelined) → IDEAL CPI > 1
Instruction Level Parallelism (ILP): So Far…

Pipeline stages

Sequential (non pipelined) IDEAL CPI > 1

Pipelining IDEAL CPI = 1
Instruction Level Parallelism (ILP): So Far…

Sequential (non pipelined) → IDEAL CPI > 1

Pipelining

Dynamic Scheduling

IDEAL CPI = 1

Single-issue out-of-order execution
Instruction Level Parallelism (ILP): So Far…

Multiple-issue out-of-order execution

IDEAL CPI < 1
(e.g. 1/3 in this case)

Superscalar

Dynamic Scheduling

Pipelining

Sequential (non pipelined)

IDEAL CPI > 1
Superscalar Execution

• This is what all high-end computers now do
  – (PowerPC, Pentium, Sparc, …)
• Main idea: why not more than one instruction beginning execution (issued) per cycle?
• Key requirements are
  – Fetching more instructions in a cycle: no big difficulty provided that the instruction cache can sustain the bandwidth
  – Decide on data and control dependencies: dynamic scheduling already takes care of this
Superscalar Processor
Dynamic Scheduler

Dynamic Scheduling:
What each unit does in each cycle is decided at execution time in hardware.

Reservation Stations, Control Logic to decide which instructions can execute, etc.

32-64 bits
Dynamic Scheduling is expensive!

- Large amount of logic, significant area cost
  - PowerPC 750 Instruction Sequencer is approx. 70% of the area of all execution units! (Integer units + Load/Store units + FP unit)

- Cycle time limited by scheduling logic (dispatcher and associated dependency checking logic)

- Design verification extremely complex
  - Very complex irregular logic
Assumptions for an ideal machine (1)

- **Register renaming** – infinite set of registers plus buffering source operands to avoid all WAW & WAR hazards
- **Branch prediction** – perfect => no mispredictions => machine with *perfect speculation* & an unbounded buffer of instructions available
- **Memory-address alias analysis** – addresses are known & a store can be moved before a load provided addresses not equal
Assumptions for an ideal machine (2)

- CPU can issue at once unlimited number of instructions per clock cycle, looking arbitrarily far ahead in computation;
- No restrictions on types of instructions that can be executed in one cycle (including loads and stores);
- \textit{One cycle latency for all instructions}; any sequence of depending instructions can issue on successive cycles;
- Instructions in execution: \textit{“in flight”}.
- \textit{Perfect caches} = all loads, stores execute in one cycle \(\Rightarrow\) only fundamental limits to ILP are taken into account.
- Obviously, results obtained are \textit{VERY} optimistic! (no such CPU can be realized…);
Perfect dynamic scheduling

• Dynamic analysis is necessary to approach perfect branch prediction (*impossible at compile time!*);

• A perfect dynamic-scheduled CPU should:

1. Look arbitrarily far ahead to find set of instructions to issue, predict all branches perfectly;

2. Rename all registers uses (⇒ no WAW, WAR hazards);

3. Determine whether there are data dependencies among instructions in the issue packet; rename if necessary;

4. Determine if memory dependencies exist among issuing instructions, handle them;

5. Provide enough replicated functional units to allow all ready instructions to issue.
Other limits of today’s CPUs

- Number of functional units
  - For instance: not more than 2 memory references per cycle
- Number of busses
- Number of ports for the register file

- All these limitations define that the maximum number of instructions that can be issued, executed or committed in the same clock cycle is much smaller than the window size
Issue-width limited in practice

- The issue width is the number of instructions that can be issued in a single cycle by a multiple issue (also called ILP) processor – And fetched, and decoded, etc.
- When superscalar was invented, 2- and rapidly 4-issue width processors were created (i.e. 4 instructions executed in a single cycle, ideal CPI = 1/4)
Now, the maximum (rare) is 6, but no more exists.

- The widths of current processors range from single-issue (ARM11, UltraSPARC-T1) through 2-issue (UltraSPARC-T2/T3, Cortex-A8 & A9, Atom, Bobcat) to 3-issue (Pentium-Pro/II/III/M, Athlon, Pentium-4, Athlon 64/Phenom, Cortex-A15) or 4-issue (UltraSPARC-III/IV, PowerPC G4e, Core 2, Core i, Core i*2, Bulldozer) or 5-issue (PowerPC G5), or even 6-issue (Itanium, but it's a VLIW).

- Because it is too hard to decide which 8, or 16, instructions can execute every cycle (too many!)
  - It takes too long to compute
  - So the frequency of the processor would have to be decreased
Summary of superscalar and dynamic scheduling

• Main advantage
  – Very high performance: Ideal CPI very low:
    \[ \text{CPI}_{\text{ideal}} = \frac{1}{\text{issue-width}} \]

• Disadvantages
  – Very expensive logic to decide dependencies and independencies, i.e. to decide which instructions can be issued every clock cycle
  – It does not scale: almost impractical to make issue-width greater than 4 (we would have to slow down the clock)
Embedded Processors

• Processors for embedded products have to be cheap and have to consume little
  – High-end processors such as 4-issue superscalar are not very suited to this: They are too expensive
  – Few superscalar processors by embedded producers

• Among the most popular embedded processors are designed by:
  – ARM (Cambridge, UK)
  – STMicroelectronics (Franco-Italian)
ARM Cortex-A8 core processor in Apple A4 System-on-Chip

- Based on the ARMv7 architecture
- It’s a dual-issue in-order execution design
- The Apple A4 at 1 GHz (45nm manufactured by Samsung from March 2010 to present), a System-on-Chip that combines an ARM Cortex-A8 and a PowerVR GPU, is in the:
  - Original iPad, April 2010
  - iPhone4: June 2010 (Black; GSM), February 2011 (Black; CDMA), April 2011 (White; GSM & CDMA)
  - iPod Touch (4th generation): September 2010 (Black model), October 2011 (White model)
  - Apple TV (2nd generation): Sept. 2010
ARM Cortex-A9 MP core processor in Apple A5 System-on-Chip

- Based on the ARMv7 architecture
- It’s a dual-issue in-order execution design
- The Apple A5 at 1 GHz (45nm to 32 nm manufactured by Samsung from March 2011 to present), a System-on-Chip that combines a dual core ARM Cortex-A9 with NEON SIMD accelerator and a dual core PowerVR GPU, is in the:
  - iPad 2 (A5 dual-core 45 nm) – March 2011; (A5 dual-core 32 nm) – March 2012
  - iPhone 4S (A5 dual-core 45 nm) – October 2011
  - Apple TV 3rd generation (A5 single-core, 32 nm) – March 2012
  - iPod Touch 5th generation (A5 dual-core 32 nm) – October 2012
  - iPad Mini (A5 dual-core 32 nm) – November 2012
Apple A6 System-on-Chip

- Apple A6 SoC was introduced on Sept. 2012 for the iPhone 5
- Apple states that it is up to twice as fast and has up to twice the graphics power compared to its predecessor the Apple A5
- The A6 uses a 1.3 GHz custom Apple-designed ARMv7 based dual-core CPU, called Swift, and an integrated triple-core PowerVR SGX 543MP3 GPU.
- The A6 chip for iPhone 5 incorporates 1GB of LPDDR2-1066 RAM and provides double the memory capacity of iPhone4S while increasing the theoretical memory bandwidth from 6.4 GB/s to 8.5 GB/s.
Power Efficiency and Performance

• They don’t always go hand in hand
• The most aggressive dynamically-scheduled superscalar have the highest performance, but consume the highest energy
  – Remember the ARM Cortex8, which is for the embedded market and it is, indeed, a much simpler core than, say, the Intel Core i7
• So let’s see some data on power vs performance
Compare two Intel Processors

- Intel i7 920
  - 4 cores, 2.66 GHz, 130 W thermal design power

- Intel Atom 230 (for the embedded market)
  - 1 core, 1.66 GHz, 4 W thermal design power

(maximum amount of power the cooling system in a computer is required to dissipate)
- Relative performance and energy efficiency
- Single-threaded benchmarks
- The i7 920 is 4 to over 10 times faster than the Atom 230 but it is about 2 times less power efficient on average
- The i7 never beats the Atom in energy
- Only one core is active on the i7 (and Atom is single-core)
What to do to avoid dynamic scheduling costs

• But to keep high performance?
• Go towards *compile-time scheduling (static scheduling):* Why not let the compiler decide which instructions can execute in parallel at every cycle?
• Instead of run-time (dynamic) scheduling…. 
Very Long Instruction Word: An Alternative Way of Extracting ILP

VLIW

Sequential (no pipelining)

Pipelining
(Statically Scheduled) Very Long Instruction Word Processor (VLIW)

Static Scheduling:
What each unit does in each cycle is decided at compile time in software
Superscalar vs VLIW Scheduling

- Deciding *when* and *where* to execute an instruction – i.e. in which cycle and in which functional unit
- For a superscalar processor it is decided at *run-time*, by custom logic in HW
- For a VLIW processor it is decided at *compile-time*, by the compiler, and therefore by a SW program
  - Good for embedded processors: Simpler HW design (no dynamic scheduler), smaller area and cheap
Challenges for VLIW

• Compiler technology
  – The compiler needs to find a lot of parallelism in order to keep the multiple functional units of the processors busy

• Binary incompatibility
  – Consequence of the larger exposure of the microarchitecture (= implementation choices) at the compiler in the generated code
### Advantages of SW vs HW Scheduling

<table>
<thead>
<tr>
<th>SW</th>
<th>HW</th>
</tr>
</thead>
<tbody>
<tr>
<td>(= Static = Complier)</td>
<td>(= Dynamic = Instruction Scheduler)</td>
</tr>
</tbody>
</table>

1) Source code available (higher level information)

2) Global analysis possible (inter-procedural analysis, etc.)

3) More time available (not bound by cycle-time)

1) Run-time information available (actual data, addresses, pointers, etc.)
Current Superscalar & VLIW processors

• Dynamically-scheduled superscalar processors are the commercial state-of-the-art for general purpose: current implementations of Intel Core i, PowerPC, Alpha, MIPS, SPARC, etc. are all superscalar

• VLIW processors are primarily successful as embedded media processors for consumer electronic devices (embedded):
  – TriMedia media processors by NXP
  – The C6000 DSP family by Texas Instruments
  – The ST200 family by STMicroelectronics
  – The SHARC DSP by Analog Devices
  – Itanium 2 is the only general purpose VLIW, a ‘hybrid’ VLIW (EPIC, Explicitly Parallel Instructions Computing)
## Taxonomy of Multiple Issue Machines

<table>
<thead>
<tr>
<th>Common name</th>
<th>Issue structure</th>
<th>Hazard detection</th>
<th>Scheduling</th>
<th>Distinguishing characteristic</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superscalar (static)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Mostly in the embedded space: MIPS and ARM, including the ARM Coretex A8</td>
</tr>
<tr>
<td>Superscalar (dynamic)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>None at the present</td>
</tr>
<tr>
<td>Superscalar (speculative)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Intel Core i3, i5, i7; AMD Phenom; IBM Power 7</td>
</tr>
<tr>
<td>VLIW/LIW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Most examples are in signal processing, such as the TI C6x</td>
</tr>
<tr>
<td>EPIC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Itanium</td>
</tr>
</tbody>
</table>
## Taxonomy of Multiple Issue Machines

<table>
<thead>
<tr>
<th>Common name</th>
<th>Issue structure</th>
<th>Hazard detection</th>
<th>Scheduling</th>
<th>Distinguishing characteristic</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superscalar</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Static</td>
<td>In-order execution</td>
<td>Mostly in the embedded space: MIPS and ARM, including the ARM Coretex A8</td>
</tr>
<tr>
<td>(static)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Superscalar</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Dynamic</td>
<td>Some out-of-order execution, but no speculation</td>
<td>None at the present</td>
</tr>
<tr>
<td>(dynamic)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Superscalar</td>
<td>Dynamic</td>
<td>Hardware</td>
<td>Dynamic with</td>
<td>Out-of-order execution with speculation</td>
<td>Intel Core i3, i5, i7; AMD Phenom; IBM Power 7</td>
</tr>
<tr>
<td>(speculative)</td>
<td></td>
<td></td>
<td>speculation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VLIW/LIW</td>
<td>Static</td>
<td>Primarily</td>
<td>Static</td>
<td>All hazards determined and indicated by compiler (often implicitly)</td>
<td>Most examples are in signal processing, such as the TI C6x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>software</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EPIC</td>
<td>Primarily static</td>
<td>Primarily</td>
<td>Mostly static</td>
<td>All hazards determined and indicated explicitly by the compiler</td>
<td>Itanium</td>
</tr>
<tr>
<td></td>
<td></td>
<td>software</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What we have learnt so far

• What is static scheduling as opposed to dynamic scheduling
• What are their advantages and disadvantages
• What can HW do better than SW in scheduling, and what can SW do better than HW
• Which processors implement which strategy and why
• What compiler techniques can be used to increase ILP

• References:
  – HP-QA 4th edition Chapter 2 (especially sec 2.7)
  – HP-QA 4th edition Appendix G
BEYOND ILP

- Multithreading (Thread-Level Parallelism, TLP)

- Multiprocessing
Yet another way to push performance

- Out-of-order multiple-issue dynamically scheduled processors: Aggressive superscalars
- Simpler core, statically scheduled, push parallelism detection towards the compiler: VLIW and EPIC processors
- Main limitation: degree of intrinsic parallelism in the instruction stream, i.e. limited amount of ILP to be exploited.

→ Simpler core, to exploit Thread-Level Parallelism instead of ILP
  - Simple multithreading such as the Sun Niagara T1
Performance beyond single thread ILP

• There can be much higher intrinsic parallelism in some applications (e.g., database or scientific codes)

• Explicit **Thread Level Parallelism** or **Data Level Parallelism**

• **Thread**: process with own instructions and data
  – thread may be a process part of a parallel program of multiple processes, or it may be an independent program
  – Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute

• **Data Level Parallelism**: Perform identical operations on data, and lots of data
Molteplicity process/thread

one process
one thread

one process
multiple threads

multiple processes
one thread per process

multiple processes
multiple threads per process

\{ = instruction trace
Threads

• Threads are created by the programmer, or by the OS
• Amount of computation assigned to each thread
  = grain size
  – Can be from a few instructions (more suited to uniprocessor multithreading: for 1 processor, need \( n \) threads)
  – To hundreds, or thousands of instructions
    (multiprocessor multithreading: for \( n \) processors, need \( n \) threads)
Threads:

Independent sequences of instructions

Single-threaded program

for (i=0; i<100; i++){
    D[i]=B[i]+A[i];
}
Threads:

Independent sequences of instructions

Single-threaded program

Multi-threaded program
Multithreading

Running more than one thread in parallel

Fetch and execute from multiple threads

Processor pipeline

To keep the execution units busy beyond ILP
What additional HW is needed in a multithreaded processor

• The processor switches between different threads → when one stalls, another goes into execution
• The state of each thread must therefore be preserved while the processor switches
• → Need multiple Register Files and multiple PCs (Program Counter registers)
• Other parts, such as for example the functional units, are not duplicated
Multithreading support

• Multithreading allows multiple threads to share the functional units of a single processor → the processor must duplicate the independent state of each thread: separate copy of register set and separate PC for each thread.

• The memory address space can be shared through the virtual memory mechanism

• The HW must support the ability to change to a different thread relatively quickly (more efficiently than a process context switch).
Exploiting TLP and ILP

• Several different flavors of multithreading for a superscalar processor:
  – **Coarse-grained Multithreading:** when a thread is stalled, perhaps for a cache miss, another thread can be executed;
  – **Fine-grained Multithreading:** switching from one thread to another thread on each instruction;
  – **Simultaneous Multithreading:** multiple threads are using the multiple issue slots in a single clock cycle.

• The Sun T1 and T2 (aka Niagara) processors are fine-grained multithreaded processors, while the Intel Core i7 and IBM Power7 processors use SMT
How the four different approaches use the execution slots of a superscalar processor. The horizontal dimension represents the instruction issue slots at each clock cycle. The vertical dimension represents the sequence of clock cycles. An empty (white) box indicates that the corresponding execution slot is unused in that clock cycle. The shades of gray and black correspond to four different threads in the multithreading processors. Black is also used to indicate the occupied issue slots in the case of the superscalar without multithreading.
Superscalar with no multithreading
Superscalar with no multithreading

• The use of issue slots is limited by a lack of ILP
• A long stall, such as an instruction cache miss, can leave the entire processor idle
• Multiple-issue processors often have more functional units parallelism available than a single thread can effectively use by ILP.
Coarse-grained Multithreading
Coarse-grained Multithreading

- Long stalls (such as L2 cache misses) are hidden by switching to another thread that uses the resources of the processor.
- This reduces the number of idle cycles, but:
  - Within each clock, ILP limitations still lead to empty issue slots
  - When there is one stall, it is necessary to empty the pipeline before starting the new thread;
  - The new thread has a pipeline start-up period with some idle cycles remaining and loss of throughput
  - Because of this start-up overhead, coarse-grained MT is better for reducing penalty of high cost stalls, where pipeline refill $\ll$ stall time
Fine-grained Multithreading
Fine-grained Multithreading
Fine-grained Multithreading

• Fine-grained MT switches between threads on each instruction → execution of multiple thread is interleaved in a round-robin fashion, skipping any thread that is stalled at that time eliminating fully empty slots.
  – The processor must be able to switch threads on every cycle.
  – It can hide both short and long stalls, since instructions from other threads are executed when one thread stalls.
  – It slows down the execution of individual threads, since a thread that is ready to execute without stalls will be delayed by another threads.
  – Within each clock, ILP limitations still lead to empty issue slots
Example of Fine-Grained MT: Sun Niagara T1

- Very simple single-issue processor core that can handle 4 threads
- There are 8 cores on the die and each core can handle 4 threads \( \rightarrow \) up to 32 threads
- Fine-Grain Multithreading: each cycle each processor core can switch between the 4 threads
Example: Architecture of SUN Niagara T1
Example: Sun Niagara T1

- Whenever a thread is stalled (for dependence, for a cache miss etc), the core executes another thread.
- Only when all four threads are stalled, the core is stalled; otherwise, it’s always busy.
- Simple, single-issue pipeline, but multithreaded.
Multithreading to hide latency

• In practice, MT is an effective way to hide long-latency events in a processor and keep the execution units busy
  ➔ If each thread is visited every (e.g.) 4 cycles, then the dependent instruction have 4 ‘free’ cycles before the dependence is resolved
T1 Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Per-thread CPI</th>
<th>Per-core CPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPC-C</td>
<td>7.2</td>
<td>1.80</td>
</tr>
<tr>
<td>SPECJBB</td>
<td>5.6</td>
<td>1.40</td>
</tr>
<tr>
<td>SPECWeb99</td>
<td>6.6</td>
<td>1.65</td>
</tr>
</tbody>
</table>

Ideal per-thread CPI = 4
(each thread ideally produces a new result every cycle, and there are 4 threads being served)

Ideal per-core CPI = 1
(it’s a single issue core)
Discussion on Performance

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Per-thread CPI</th>
<th>Per-core CPI</th>
<th>Effective CPI for 8 cores</th>
<th>Effective IPC for 8 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPC-C</td>
<td>7.2</td>
<td>1.80</td>
<td>0.225</td>
<td>4.4</td>
</tr>
<tr>
<td>SPECJBB</td>
<td>5.6</td>
<td>1.40</td>
<td>0.175</td>
<td>5.7</td>
</tr>
<tr>
<td>SPECWeb99</td>
<td>6.6</td>
<td>1.65</td>
<td>0.206</td>
<td>4.8</td>
</tr>
</tbody>
</table>

1.8 CPI does not sound so high compared to an aggressive dynamically scheduled superscalar ILP core (ideal CPI = ¼ …)

**BUT:**

Since the core is really simple (single-issue!), T1 can pack 8 cores in a chip, while the more aggressive ILP ones could have only 2 to 4 cores in a chip

As a result, performance is comparable, if not better
Simultaneous Multithreading (SMT)

Why not to do both TLP and ILP?
Simultaneous Multithreading (SMT)
Simultaneous Multithreading (SMT)

- TLP and ILP are exploited simultaneously → multiple thread are using the multiple issue slots in a single clock cycle.
- Resolution of multiple dependences can be handled by the dynamic scheduling capabilities of the processor.
- Register renaming provides unique register identifiers to mix instructions from multiple threads in the data-path without confusing sources/destinations across threads.
- Issue slots usage is still limited by imbalances of resource needs/availability over multiple threads.
- Mixing many threads might inevitably compromise the execution time of individual threads.
SMT

• Key motivation: a CPU today has more functional resources that what one thread can actually use
• Simultaneously schedule instructions for execution from all threads
• It’s the most common implementation of multithreading today: Intel Core i7, IBM Power7
• It arises naturally when Fine MT is implemented on top of a multiple-issue, dynamically-scheduled processor
Exploit more parallelism than Fine MT

Fine MT makes sense in small-issue machines (1 or 2-issue)

In fact, T1 Niagara is a fine-MT single-issue:

But in a large-issue machine (4-issue), where there are many slots to keep busy, SMT is the most obvious choice: let instructions from different threads be executed in the same cycle.
Performance comparison

Number of threads
SMT performance

- The speedup from using multithreading on one core on an i7 processor
- The speedup averages 1.28 for the Java benchmarks and 1.31 for the PARSEC benchmarks.
- The energy efficiency averages 0.99 and 1.07, respectively.
- A value of energy efficiency above 1.0 for energy efficiency indicates that the feature reduces execution time by more than it increases average power.
- These data were collected and analyzed by Esmaeilzadeh et al. [2011]
Improved Fetch Unit

• The Fetch Unit in a SMT architecture has two distinct advantages over a conventional architecture:
  ✓ Can fetch from multiple threads at once
  ✓ Can choose which threads to fetch
• Fetching from 2 threads/cycle achieves most of the performance from multiple-thread fetch.
• How to identify the best candidate to fetch?
• Advantages:
  ✓ Fewest unresolved branches
  ✓ Fewest load misses
  ✓ Fewest instructions in queue
SMT

- The threads in an SMT design are all sharing just one processor core, and just one set of caches, has major performance downsides compared to a true multiprocessor (or multi-core).
- On the other hand, applications which are limited primarily by memory latency (but not memory bandwidth), such as database systems and 3D graphics rendering, benefit dramatically from SMT, since it offers an effective way of using the otherwise idle time during cache misses.
- Thus, SMT presents a very complex and application-specific performance scenario.
Multicore and SMT

• Multiple cores where each processor can use SMT
• Number of threads: 2, 4 or sometime 8
  • (called “hyperthreading” by Intel)
• Memory hierarchy:
  – If only multithreading: all caches are shared
  – Multicore:
    • Cache L1 private
    • Cache L2 private in some architectures and shared in others
    • Memory always shared
SMT Dual core: 4 concurrent threads
SMT

- **Intel Pentium-4** was the first processor to use SMT, which Intel calls "hyper-threading", supporting 2 simultaneous threads. **Intel's Core i** and **Core i*2** are also 2-thread SMT, as is the low-power **Atom x86** processor. A typical quad-core **Intel Core i7** processor (max freq. 3.7 GHz) is thus an 8 thread chip.

- **IBM POWER 7** superscalar symmetric multiprocessor (45nm, up to 4.25 GHz) has up to 8 cores, and 4 threads per core, for a total capacity of 32 simultaneous threads (SMT).

- Sun was the most aggressive of all on the TLP front, with **UltraSPARC-T1** (aka: "Niagara") providing 8 simple in-order cores each with 4-thread, for a total of 32 threads on a single chip. This was subsequently increased to 8 threads per core in **UltraSPARC-T2**, and then 16 cores in **UltraSPARC-T3**, up to 128 threads!
Funny Quote

• “If you were plowing a field, which would you rather use: two strong oxen or 1024 chickens?”

  By Seymour Cray, father of the supercomputer (arguing for two powerful vector processors vs. many simple processors)

• Many and simple cores, or few and wider cores?

  Although there isn’t a “right” answer, my research is oriented to investigate about multi- and many-cores.
Important take-home point

Multithreading reduces execution time by more than it increases average power.

Aggressive ILP techniques, instead, such as speculation and predicated execution, don’t have the above important characteristic.