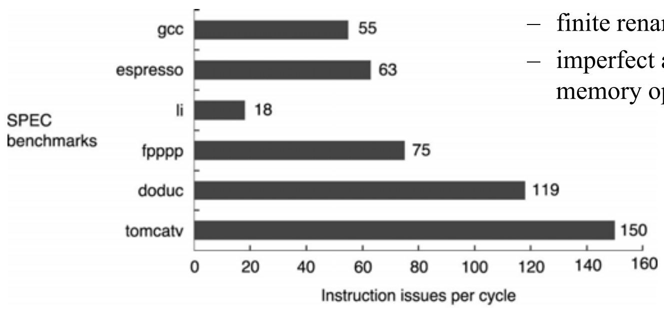
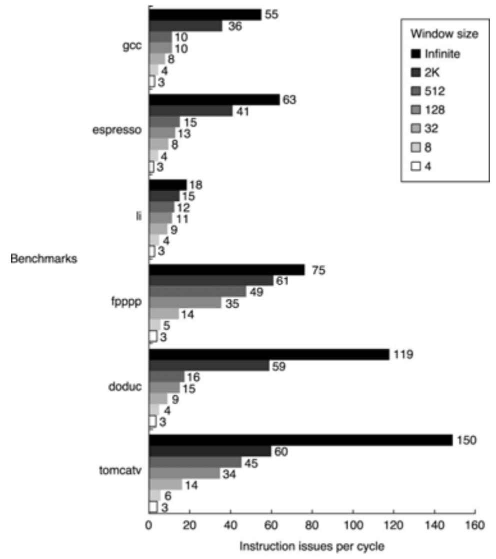
ILP in real code

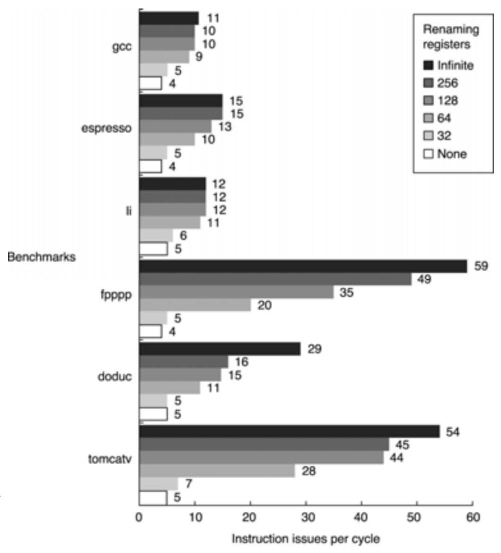


- Based on all kinds of ideal assumptions. Further limited by:
  - realistic branch prediction
  - finite renaming registers
  - imperfect alias analysis for memory operations

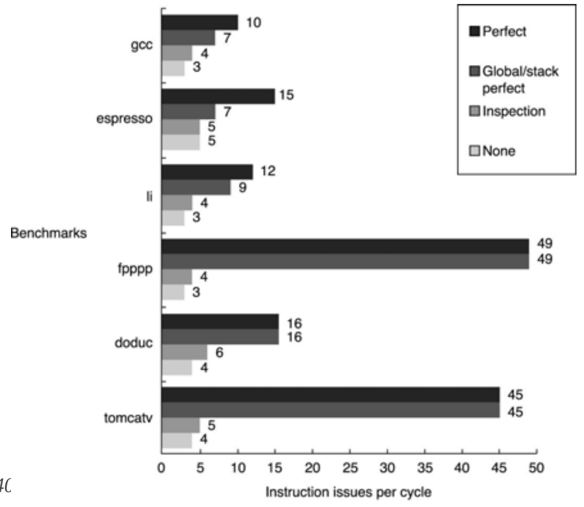
Window Size



Renaming Registers



Memory Aliasing

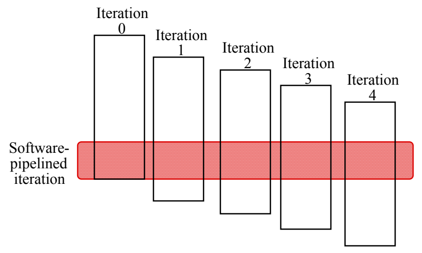


Exposing More ILP

- These techniques were originally motivated by VLIW, which needs tons of ILP to work at all – but useful for superscalar/dynamic/speculative processors, as well.
- Software Techniques
  - Software Pipelining
  - Trace Scheduling
- Hardware/Software Technique
  - Predicated execution
  - Simultaneous Multithreading

Compiler support for ILP: Software Pipelining

- Observation: if iterations from loops are independent, then can get ILP by taking instructions from different iterations
- Software pipelining: reorganizes loops so that each iteration is made from instructions chosen from different iterations of the original loop



SW Pipelining Example

Unrolled 3 times

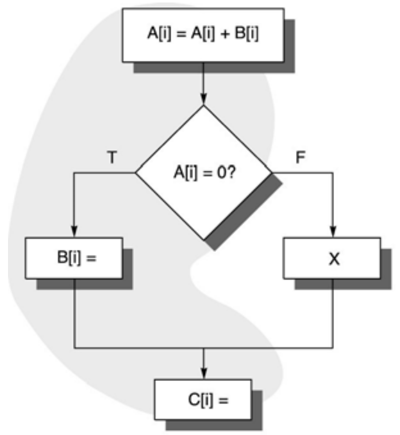
```
1   LD    F0,0(R1)
2   ADDD  F4,F0,F2
3   SD    0(R1),F4
4   LD    F6,-8(R1)
5   ADDD  F8,F6,F2
6   SD    -8(R1),F8
7   LD    F10,-16(R1)
8   ADDD  F12,F10,F2
9   SD    -16(R1),F12
10  SUBI  R1,R1,#24
11  BNEZ  R1,LOOP
```

Software Pipelined

```
    LD    F0,0(R1)
    ADDD  F4,F0,F2
    LD    F0,-8(R1)
1LP:SD    0(R1),F4;     Stores M[i]
2   ADDD  F4,F0,F2;     Adds to M[i-1]
3   LD    F0,-16(R1);   loads M[i-2]
4   SUBI  R1,R1,#8
5   BNEZ  R1,LP
    SD    0(R1),F4
    ADDD  F4,F0,F2
    SD    -8(R1),F4
```

Compiler Support for ILP: Trace Scheduling

- Creates long basic blocks by finding long paths in the code



Trace Scheduling

- Parallelism across IF branches vs. LOOP branches
- Two steps:
  - *Trace Selection*
    - Find likely sequence of basic blocks (*trace*) of (statically predicted) long sequence of straight-line code
  - *Trace Compaction*
    - Squeeze trace into few VLIW instructions
    - Need bookkeeping code in case prediction is wrong

HW Support for More ILP

- Predication – Trading off branch hazards and control flow constraints for increased instruction bandwidth
- Case Studies
- Simultaneous Multithreading – Transforming thread level parallelism (TLP) into ILP

Predication: HW support for More ILP

- Avoid branch prediction by turning branches into *conditionally executed instructions*: (aka predicated instructions)

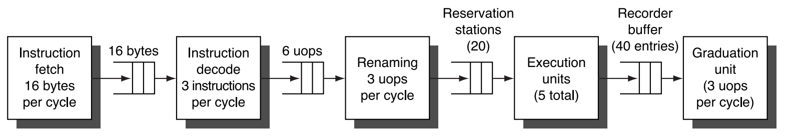
  add c, a, b (x) => if (x) then a = b + c else NOP
  - If false, then neither store result nor cause exception
  - Expanded ISA of Alpha, MIPS, PowerPC, SPARC have conditional move; PA-RISC can annul any following instr, IA64 can predicate any instruction (even have multiple predicates)

```
        ld     F2, 0(R2)
        addd   F4, F2, F0
        multd  F6, F4, F4
        beqz   R3, go_on
        addd   F10, F0, F8
        addi   R2, R2, #8
go_on:  addi   R2, R2, #8
        bnez   F6, loop
```

Predicated Execution

- Drawbacks to conditional instructions
  - Still takes a clock & alu even if "annulled"
  - Stall if condition evaluated late
  - Complex conditions reduce effectiveness; condition becomes known late in pipeline
  - requires more operands! Typically only available as *conditional move*.
- Advantages
  - eliminate prediction, misprediction
  - longer basic blocks, ...
- Critical technology for VLIW, sw pipelining. Why?

Pentium Pro (II, III, etc.) microarchitecture


Instruction fetch 16 bytes per cycle
16 bytes
Instruction decode 3 instructions per cycle
6 uops
Renaming 3 uops per cycle
Reservation stations (20)
Execution units (5 total)
Recorder buffer (40 entries)
Graduation unit (3 uops per cycle)

- 40 *uncommitted* instructions
- 20 *unissued* instructions

Pentium 4 microarchitecture

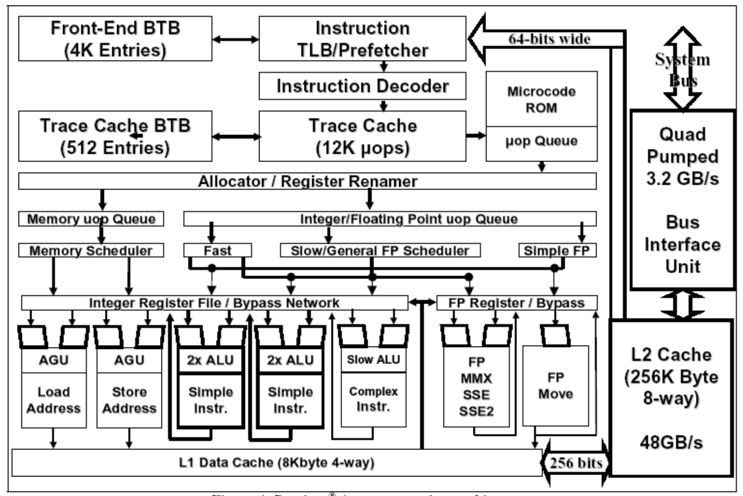
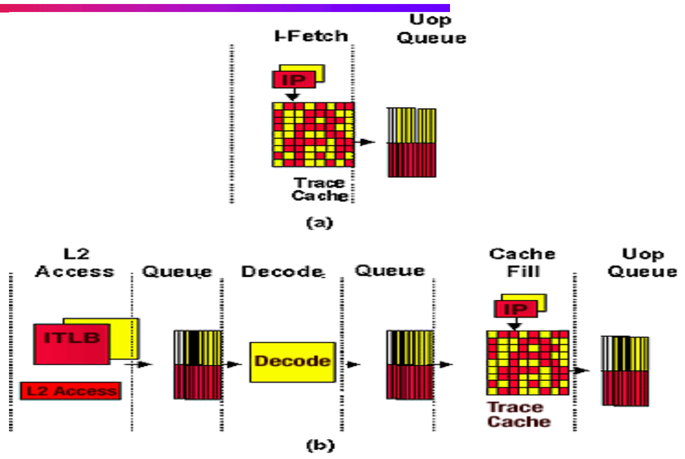
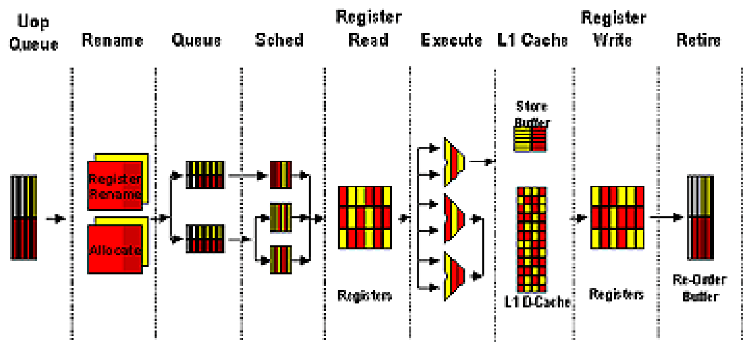


Figure 4: Pentium® 4 processor microarchitecture

Pentium 4 front-end



Pentium 4 back-end



- *126 in-flight* instructions (ROB size)

Pentium 4 Summary

- 20-stage pipeline
- IA32 (x86) ISA translated to RISC-like uops
- Uops stored in trace cache
- Decode/retire 3 uops/cycle
- Execute 6 uops/cycle
- Dynamically scheduled
- Explicit Register Renaming
- Simultaneous Multithreading (hyper-threading)

Intel Nehalem (Core I7)

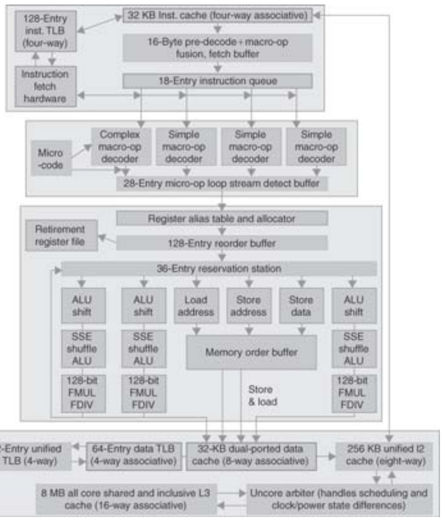

128-Entry inst. TLB (four-way)
32 KB Inst. cache (four-way associative)
16-Byte pre-decode + macro-op fusion, fetch buffer
Instruction fetch hardware
18-Entry instruction queue
Micro -code
Complex macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
Simple macro-op decoder
28-Entry micro-op loop stream detect buffer
Register alias table and allocator
Retirement register file
128-Entry reorder buffer
36-Entry reservation station
ALU shift
ALU shift
Load address
Store address
Store data
ALU shift
SSE shuffle ALU
SSE shuffle ALU
Memory order buffer
SSE shuffle ALU
128-bit FMUL FDIV
128-bit FMUL FDIV
Store & load
128-bit FMUL FDIV
512-Entry unified L2 TLB (4-way)
64-Entry data TLB (4-way associative)
32-KB dual-ported data cache (8-way associative)
256 KB unified l2 cache (eight-way)
8 MB all core shared and inclusive L3 cache (16-way associative)
Uncore arbiter (handles scheduling and clock/power state differences)

Figure 3.41 The Intel Core i7 pipeline structure shown with the memory system components. The total pipeline depth is 14 stages, with branch mispredictions costing 17 cycles. There are 48 load and 32 store buffers. The six independent functional units can each begin execution of a ready micro-op in the same cycle.

ILP Summary

- Parallelism is absolutely critical to modern computer system performance, but at a very fine level.
- Mechanisms that create, or expose parallelism: loop unrolling, software pipelining, code motion
- Mechanisms that allow the machine to exploit ILP: pipelining, superscalar, dynamic scheduling, speculative execution

Simultaneous Multithreading

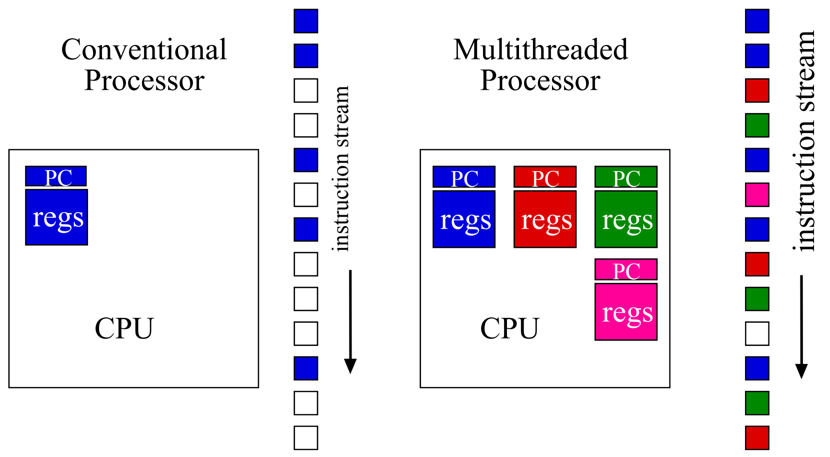
Tullsen, Eggers, Levy, *Simultaneous Multithreading: Maximizing On-Chip Parallelism*, ISCA, 1995

Tullsen, Eggers, Emer, Levy, Lo, Stamm, *Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor*, ISCA, 1996

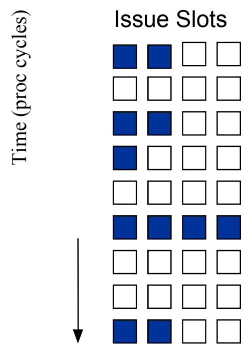
Motivation

- Modern processors fail to utilize execution resources well.
- There is no single culprit.
- Attacking the problems one at a time (e.g., *specific* latency-tolerance solutions) always has limited effectiveness.
- However, a *general latency-tolerance solution* which can hide all sources of latency can have a large impact on performance.

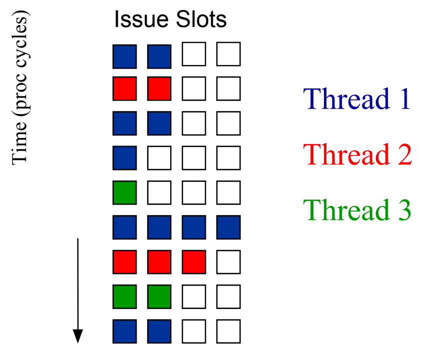
Hardware Multithreading



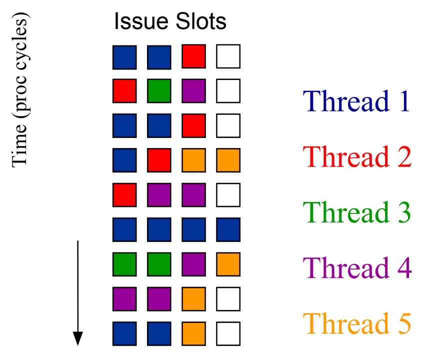
Superscalar Execution



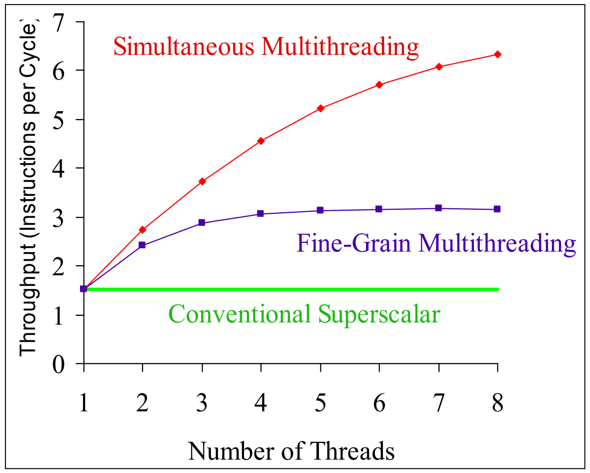
Superscalar Execution with Fine-Grain Multithreading



Simultaneous Multithreading



The Potential for SMT

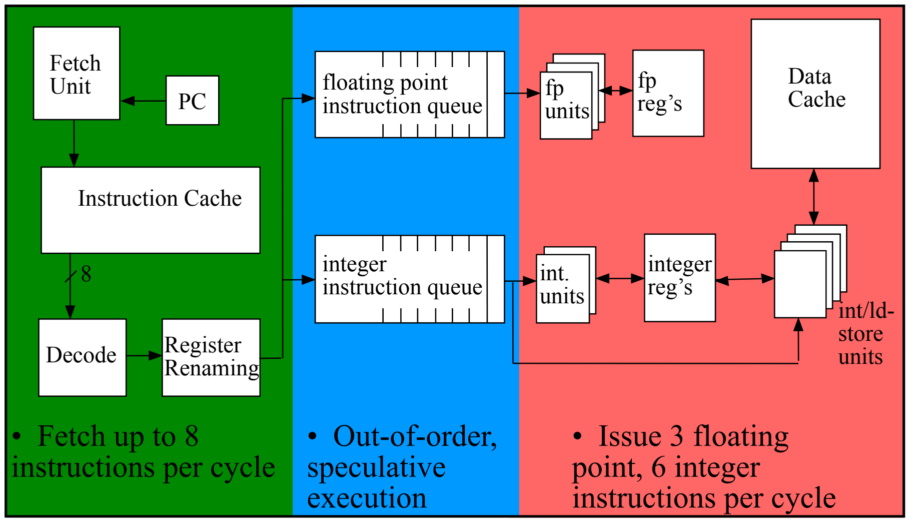


Goals

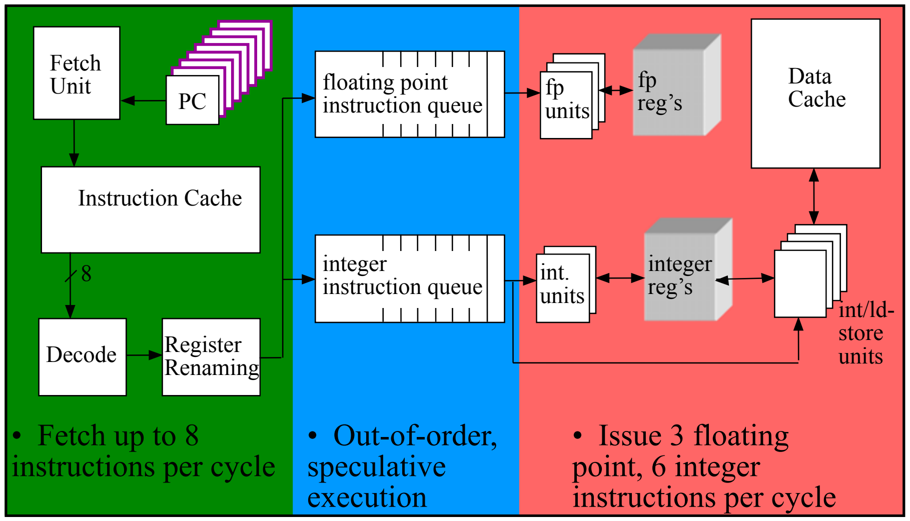
We had three primary goals for this architecture:

1. Minimize the architectural impact on conventional superscalar design.
2. Minimize the performance impact on a single thread.
3. Achieve significant throughput gains with many threads.

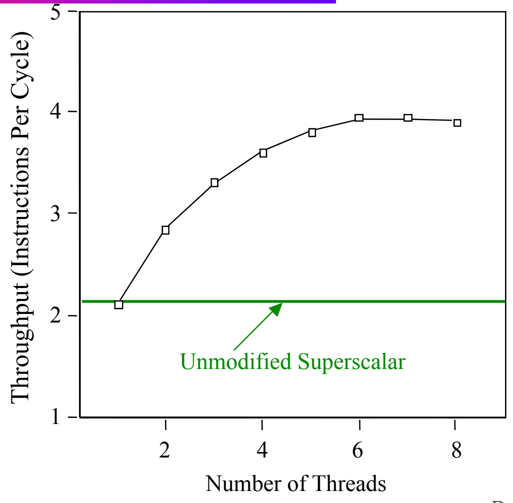
A Conventional Superscalar Architecture


Fetch Unit
PC
Instruction Cache
8
Decode
Register Renaming
floating point instruction queue
integer instruction queue
fp units
fp reg's
int. units
integer reg's
Data Cache
int/ld-store units
• Fetch up to 8 instructions per cycle
• Out-of-order, speculative execution
• Issue 3 floating point, 6 integer instructions per cycle

An SMT Architecture


Fetch Unit
PC
Instruction Cache
8
Decode
Register Renaming
floating point instruction queue
integer instruction queue
fp units
fp reg's
int. units
integer reg's
Data Cache
int/ld-store units
• Fetch up to 8 instructions per cycle
• Out-of-order, speculative execution
• Issue 3 floating point, 6 integer instructions per cycle

Performance of the Naïve Design



Bottlenecks of the Baseline Architecture

- Instruction queue full conditions (12-21% of cycles)
  - Lack of parallelism in the queue.
- Fetch throughput (4.2 instructions per cycle when queue not full)

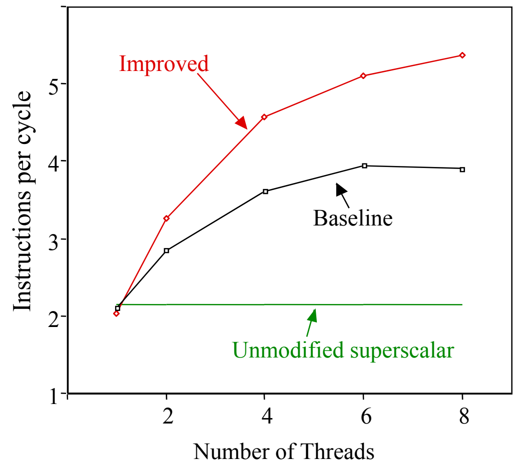
Improving Fetch Throughput

- The fetch unit in an SMT architecture has two distinct advantages over a conventional architecture.
  1. Can fetch from multiple threads at once.
  2. Can choose which threads to fetch.

Improved Fetch Performance

- Fetching from 2 threads/cycle achieved most of the performance from multiple-thread fetch.
- Fetching from the thread(s) which have the fewest unissued instructions in-flight significantly increases parallelism and throughput.

Improved Performance



This SMT Architecture, then:

- Borrows heavily from conventional superscalar design.
- Minimizes the impact on single-thread performance
- Achieves significant throughput gains over the superscalar (2.5X, up to 5.4 IPC).

Commercial SMT

- Alpha 21464 (☹)
- Intel Pentium 4 “hyper-threading” processor.
- IBM Power 5 – 2 cores, 2 SMT threads/core
- IBM Power 6 – again, 2 cores, 2 SMT threads/core
- IBM Power 7 – 8 cores, 4 SMT threads/core
- Sun Niagara T1 (2006) – 8 cores, 4 threads/core (SMT?)
- Sun Niagara T2 – 8 cores, 8 threads/core
- Intel Nehalem (core i7) 4-8 cores, 2 SMT threads/core