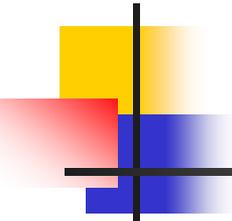


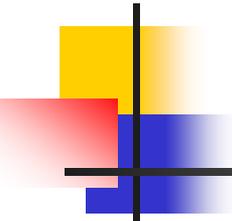
Introductory SMT!

Hassan Shojanian



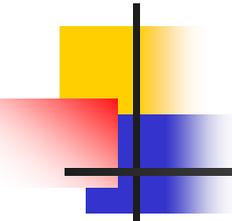
Agenda

- History [Orlowski '01]
- Basics
- “Simultaneous Multithreading: Maximizing on-chip parallelism” [Tullsen '95]
- “Exploiting choices: Instruction fetch and issue on an Implementable SMT processor” [Tullsen '96]
- Pentium4 case study [Koufaty '03] [Marr '02]
- Future



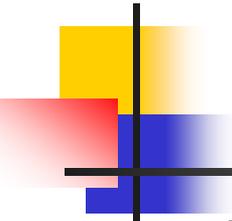
Some history

- 1959: TX-2 @ MIT used fast context switch for I/O.
- 1964: CDC 6600 I/O processor.
- 1979: HEP - First multithreaded CPU made by Burton Smith at Denelcor.
- 1988: Smith founded Tera.
 - Tera series: Fine-grained multithreaded processor
[Tera acquires Cray from SGI later (2000) and change name to Cray Inc.; now builds MTA series]
- 1994-5: Tullsen/Eggers/Levy @ Univ. of Washington were working on general model of SMT
 - More aggressive approach => SMT
 - The first paper: *SM: Maximizing on-chip parallelism*
 - Idea: *Whatever instruction from whatever thread!*
 - Joel Emer at DEC becomes interested.



Some history -2

- 1996: Follow-up paper with Joel Emer and Rebecca Stamm (also from DEC)
 - Refined the original idea; more complete architecture
 - Convinced the industry that this was not hard to do.
 - Even can be done for commodity microprocessors
- 1999: Compaq announces first mainstream multi-threaded CPU: the *Alpha EV8* (4-way SMT, 8-issue)
- 2000: Intel secretly works on its own SMT
- 2001: Compaq cancels *Alpha EV8* project
- 2001: Intel announces SMT supported Foster/Jackson (aka. Xeon/P4) for 2002
- 2004: IBM's Power5, Sun's UltraSPARC IV
- 2005(?): Intel's Tulsa (dual core SMT)



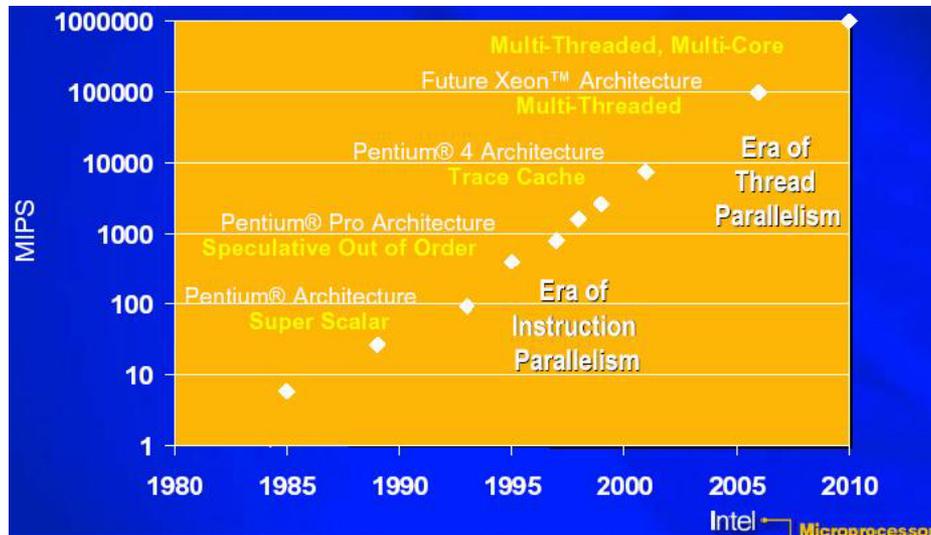
Basics

- Modern processors:
 - Employ:
 - Super-pipelining to achieve higher clock rate
 - Superscalar techniques (multi-issue, out-of-order execution on multiple functional units, in-order completion) to exploit ILP
 - Cache hierarchy to decrease latency
 - Challenge: find enough instructions to execute in parallel
 - Ultimately limited by dependencies/long latency operations
 - Higher increase of transistor count/power than performance
 - Architects look for new techniques to achieve opposite
- TLP:
 - Parallelism across multiple threads
 - Multi-threaded applications esp. servers (e.g. Transaction Processing)
 - Shared multi-tasking workloads
 - Solutions in form of:
 - Conventional multiprocessors (SMP) → expensive
 - Chip multiprocessors (CMP) → large die-size; doesn't address power
 - Multithreaded processors

Basics

-2

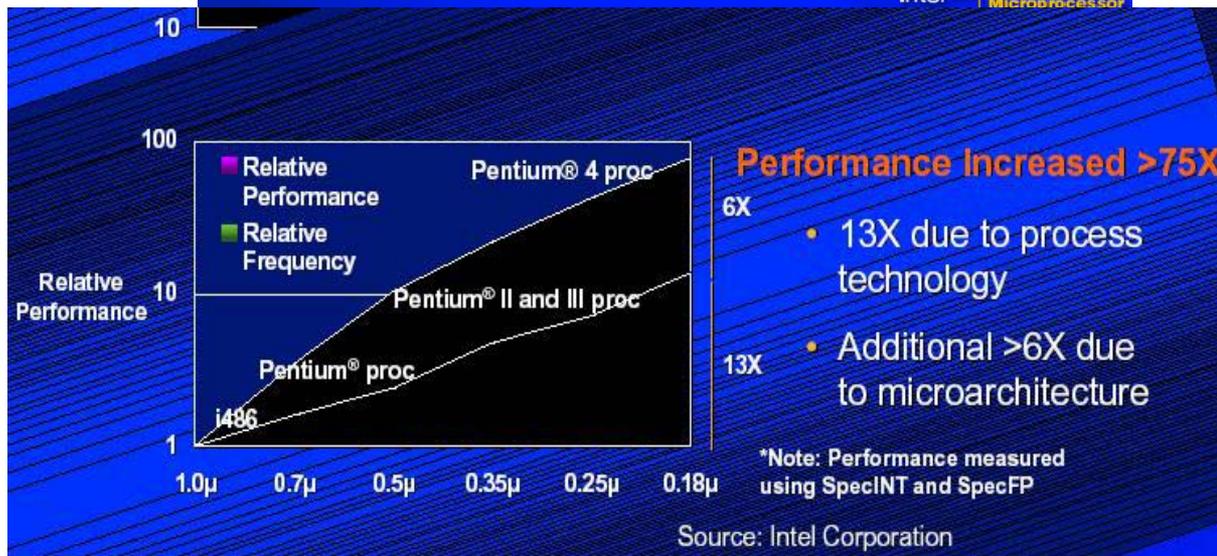
- Parallelism in transition:

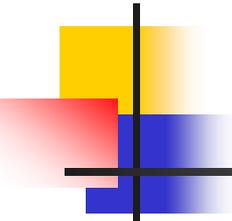


(Both from [Shen '02])

- Performance increase:

- Process technology
- Microarchitecture





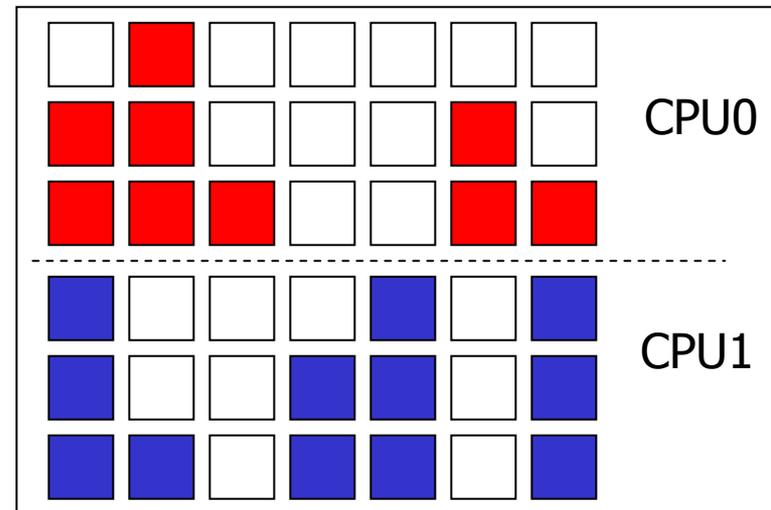
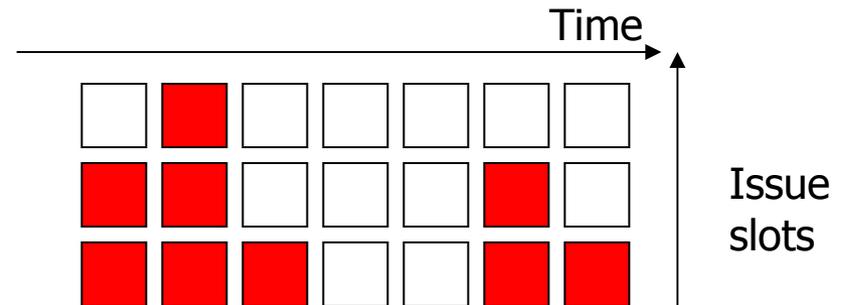
Basics -3

- Conventional processors:
 - OS binds one SW thread to a CPU core at anytime
- Multithreaded processor:
 - Exploit TLP but with a single core (against CMP)
 - Multiple architectural contexts (e.g. Reg. file)
 - Multiple virtual processors
 - OS sees multiple independent processors; multiple SW threads
 - Hides latencies by executing from another thread
 - Shared functional units
 - Is issue BW shared among multiple thread at any cycle (i.e. no architectural context switch)?
 - No: Traditional multithreaded processors
 - Yes: SMT

Basics

-4

- 1) Superscalar
 - Both horizontal (cycles with all issue slots wasted) and vertical (empty issue slots) waste exist
- 2) Chip Multiprocessor (CMP)
 - Similar under-utilization of execution units as superscalar
 - But total throughput is increased
 - Fixed thread/FU assignment

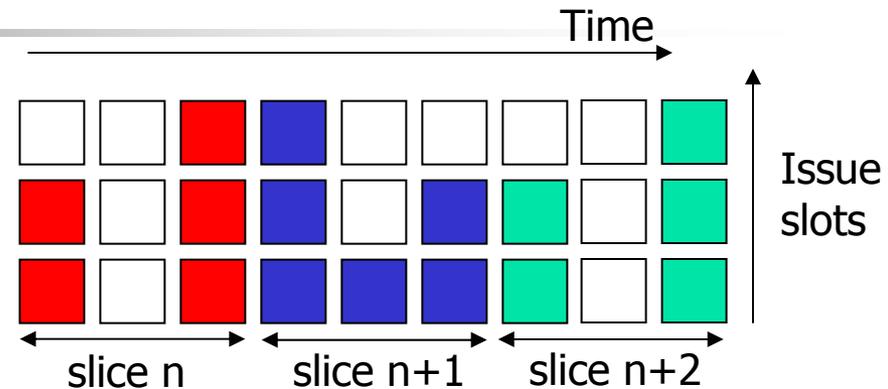


Basics

-5

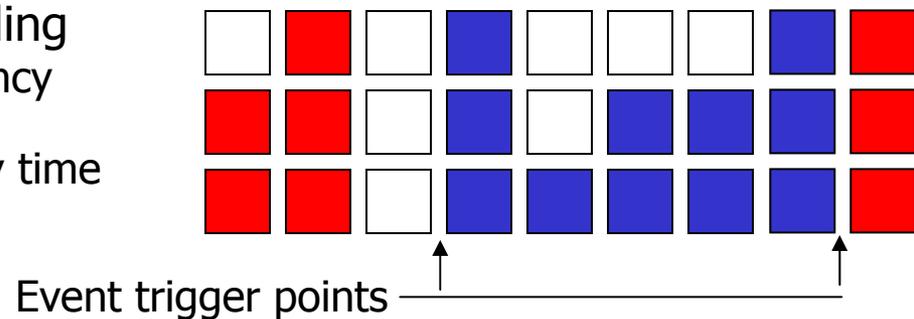
3) Time-sliced Multithreading

- Context switch at fixed time point
- Can waste issue slots
- One context (thread) active at any cycle
- Targets horizontal waste



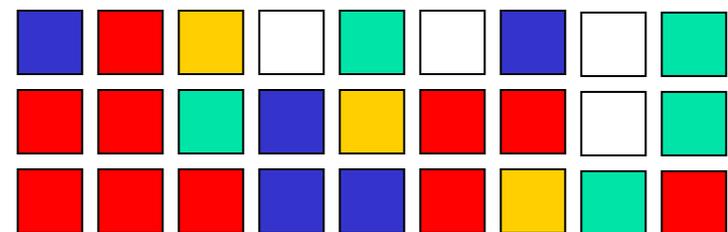
4) Switch-on-event Multithreading

- Switches threads on long-latency events such as cache misses
- Only one context active at any time
- Targets horizontal waste

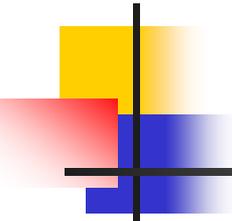


5) Simultaneous Multithreading

- All contexts are active
- Any instruction from any thread
- Targets **both** horizontal & vertical waste
- Dynamic assignment of FUs

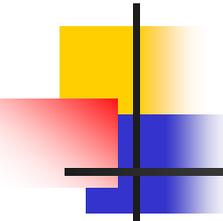


- Conventional multithreading:
 - Fine-grain multithreading
 - Interleaving at cycle level; usu. round-robin skipping stalled threads
 - **Pro:** Hides short/long stalls; **Con:** Slows down individual threads
 - Tera: Up to 128 active contexts; scheduling one at any cycle
 - HEP: Executes a thread once every 8 cycles → Poor single-threaded performance
 - Coarse-grain multithreading
 - Context switch only on costly stalls (e.g. at remote-memory access or failed synchronization attempt in APRIL)
 - **Pro:** At long stalls; **Con:** Freeze pipeline at switch → lengthy startup
- Other terms:
 - Also might see “Asynchronous/Synchronous Multithreading” to refer to “conventional/SMT”
 - CMT (Chip Multi-Threading) by Sun: SMT and/or CMP
 - General term: “A processor's ability to process multiple software threads”



SM: Maximizing...

- Paper's contributions:
 - Provides several SMT models
 - Compares their performance with superscalar, fine-grained multithreading and CMP
 - Tunes cache hierarchy of SMT model
- Weakness
 - Acknowledge optimistic approach
 - Suggests it could be upper bound result

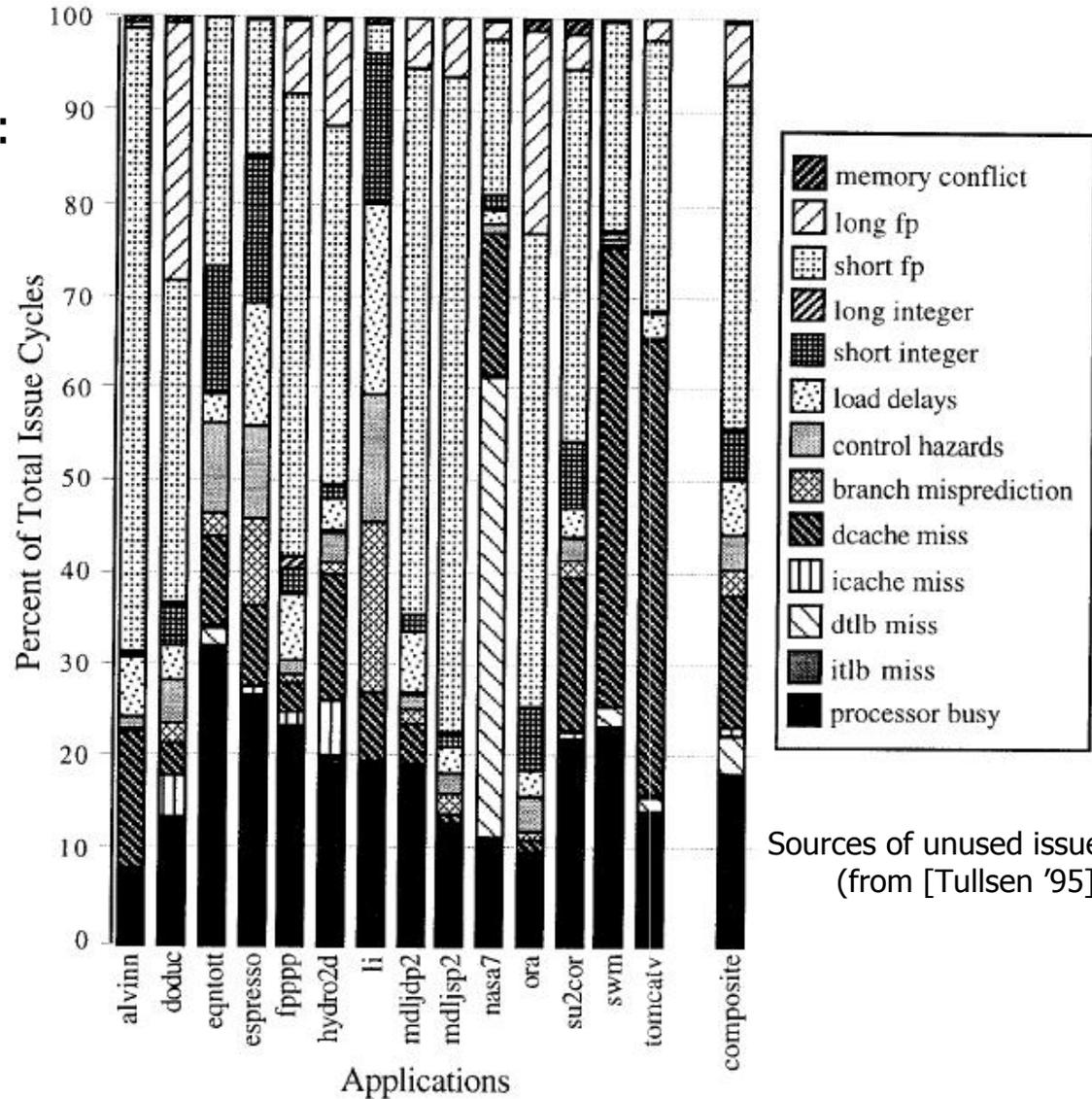


SM: Maximizing... -2

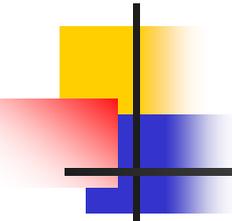
- Methodology:
 - Base architecture of Alpha 21164
 - 8-issue, 10 pipelined functional units (4int, 2fp, 3ld/st, 1br)
 - Lockup free multi-banked caches; 2K 2-bit BHT, ...
 - Limited dynamic scheduling: in-order issue of dependence free instr. to scheduling window; OOO issue to FU
 - Reduces complexity (no register renaming); scoreboarding
 - Static scheduling with Multiflow trace scheduling compiler
 - Using strict priority order for scheduling instructions from different threads
 - Workload
 - SPEC92 benchmark suite
 - Single-threaded applications; multiprogramming workload rather than parallel processing; no synch delays
 - Measured cause & percentage of empty issue slots

SM: Maximizing... -3

- Sources of unused issue slots for pure superscalar:
 - 'processor busy' shows utilized issue slots; all other slots wasted
 - No single dominant cause for all; different for applications
 - Less than 1.5 IPC
 - 61% horizontal waste (stalls)! Remaining 39% from vertical waste
 - (only *eqntott*, *espresso* & *li* are integer)



Sources of unused issue slots (from [Tullsen '95])

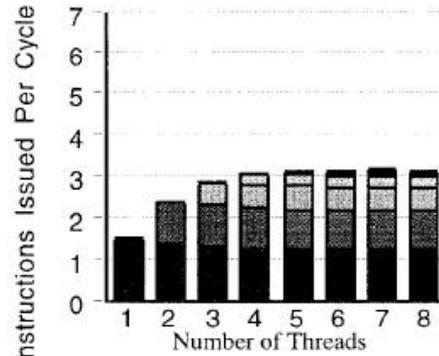


SM: Maximizing... -4

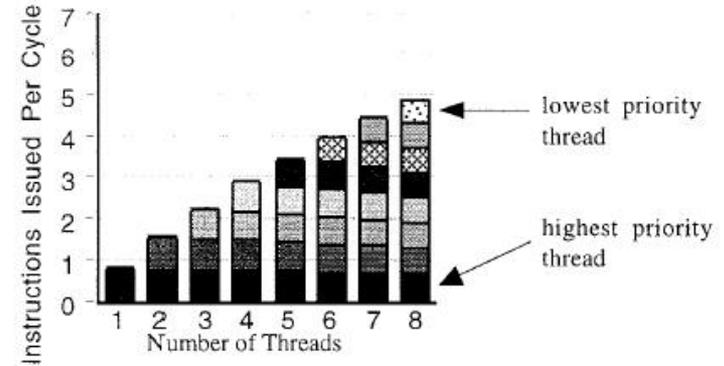
- Models:
 - Fine-grain multithreaded:
 - One thread uses the whole issue-width every cycle
 - SM-Full:
 - Full SMT
 - SM-[single/dual/four] Issue:
 - Limits maximum number of instructions issued from each thread in any cycle
 - SM-Limited Connection:
 - Functional units are shared among threads=> simpler architecture
 - Note the thread priority order at issue

SM: Maximizing... -5

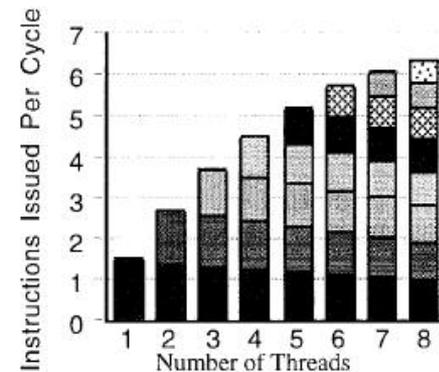
- Fine grain:
 - 3.2 IPC
 - Not much beneficial after 4 threads
 - Horizontal waste < 3%
 - Filled with other threads
- SM-Single Issue:
 - Almost equal share for each thread
- SM-Full:
 - As high as 6.3 IPC
 - increased utilization
 - SM-Four Issue achieves almost the same IPC
 - Even 2-issue is good
 - **Negative effect of sharing:** for even non-execution resources (cache, TLB, BHT, ...)
 - 35% drop in 1st thread's
- SM-Limited Connection:
 - slower increase in IPC



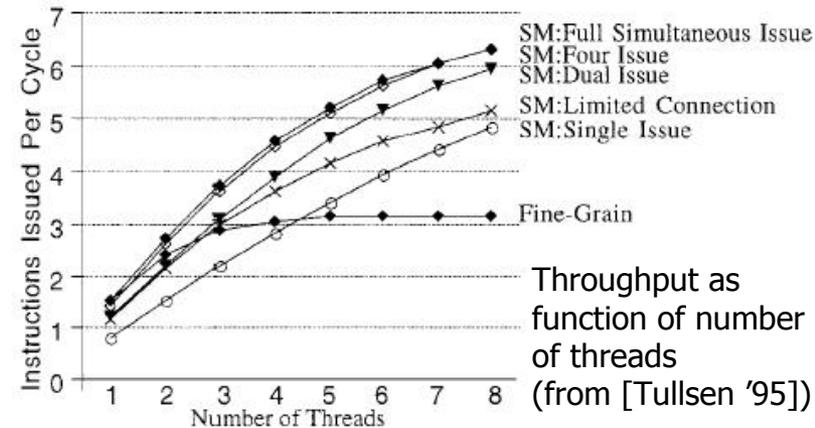
(a) Fine Grain Multithreading



(b) SM: Single Issue Per Thread



(c) SM: Full Simultaneous Issue



(d) All Models

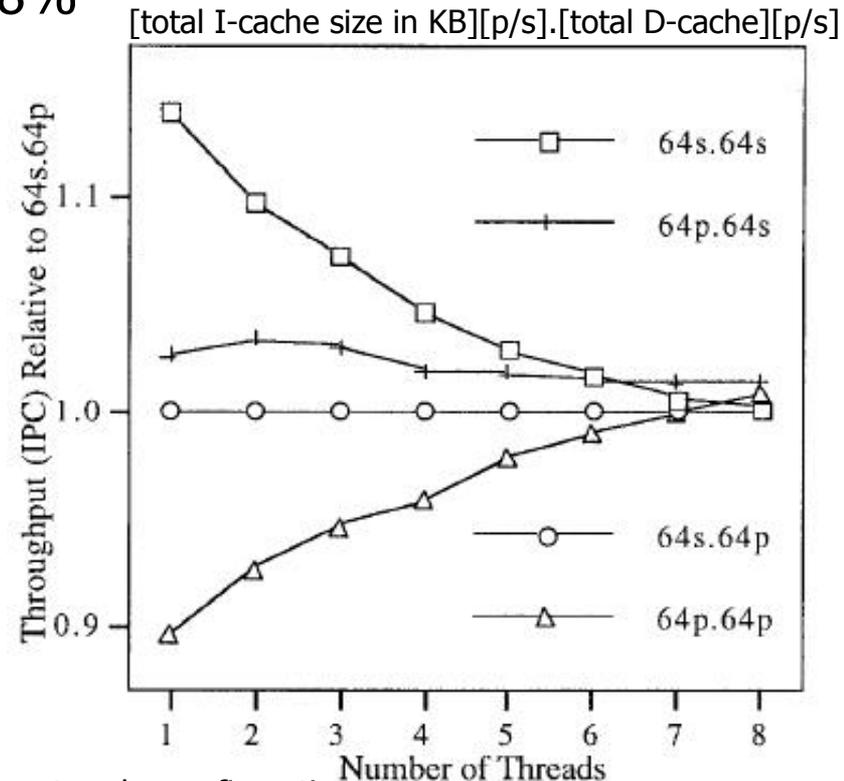
Throughput as function of number of threads (from [Tullsen '95])

- Points:
 - Can trade complexity with number of contexts at fixed IPC (watch for available SW threads)
 - Increased utilization is *direct result* of threads *dynamically* sharing processor resources
 - Sharing the same resources wasted in superscalar in idle state

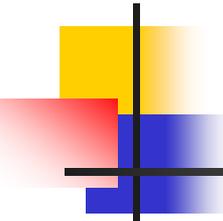
SM: Maximizing... -6

- Negative effect of sharing:
 - Dominant factor: Sharing cache
 - I-cache misses grows from 1% to 14% (from 1 to 8 threads)
 - D-cache misses from 12% to 18%
 - TLB misses from 1% to 6%

- Different cache design:
 - Private vs. shared
 - Different access pattern for I/D caches
 - Shared I-cache better for small # of threads; private better for large
 - 64s.64s: best choice with varying # of active threads
 - Also better for data-sharing

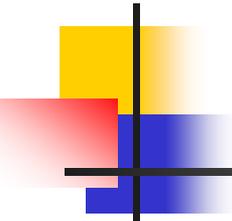


Different cache configurations
(from [Tullsen '95])



SM: Maximizing... -7

- SM vs. CMP with equal resources
 - Both: multi-issue, multi register sets, multi FUs, ...
 - Static partitioning (CMP) vs. dynamic partitioning (SMT) of issue width, FUs
 - Different experiments: (equal cache size per thread vs. per core)
 - SM: 4 thread, 8-issue; CMP: 4 2-issue
 - FU=16, Issue BW=8, Reg. sets=4: *4.15 vs. 3.44 IPC*
 - SM: 8 thread, 8-issue; CMP: 8 4-issue
 - FU=32, Issue BW=8 vs.32, Reg. sets=8: *6.64 vs. 6.35 IPC*
 - SMT's advantages:
 - Better utilization with few threads (1/n goes idle in CMP)
 - Better granularity; in smaller units not entire processor
 - In multi-threaded apps, shared D1 means no need for coherency
 - SMT's drawbacks:
 - More complex to design/verify (e.g. instruction scheduling)



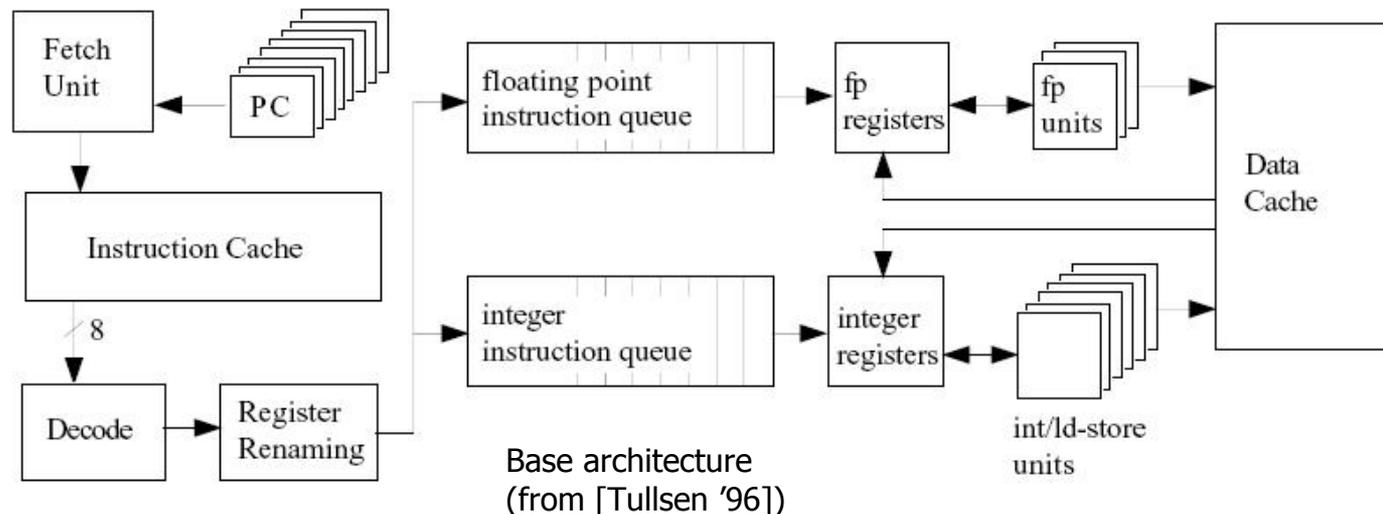
Exploiting choices... -1

- Weakness of previous paper ([Tullsen '95]):
 - Quite idealized model
 - Not detailed architecture (e.g. fetch unit)
 - Optimistic in pipeline stages (inst. scheduling, register file access, ...)
- [Tullsen '96] contributions:
 - More realistic model
 - Not extensive change required to support SMT
 - Not compromising performance of single-thread applications
 - SMT achieves high throughput gain
 - 5.4 vs. 2.1 IPC (unmodified superscalar) on 9-issue/8 fetch
 - Remember 6.3 vs. 1.5 IPC on 8-issue in previous work
 - Favoring efficient threads over slow ones in issue/fetch stages
 - pick *best* instructions from *all* threads

Exploiting choices... -2

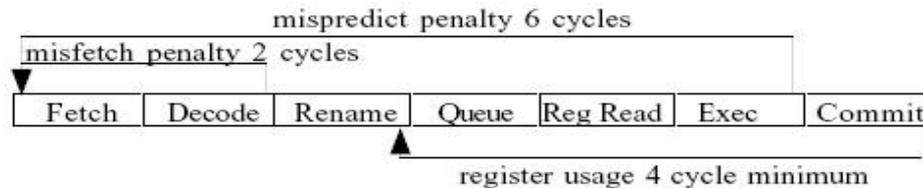
Architecture:

- Simple extension of conventional superscalar
- Fetches up to 8 instructions per cycle
- Shared instruction queue; OOO execution, in-order completion
- Register renaming: Thread transparent scheduling □
- Multiple arch. state (100 more registers considered for renaming)
- Some partitioning: retirement, queue flush, return stack □
- First step: fine-grain at fetch (round-robin), SMT at execution □
- Using a thread ID to identify some elements (in IQ, BTB) □



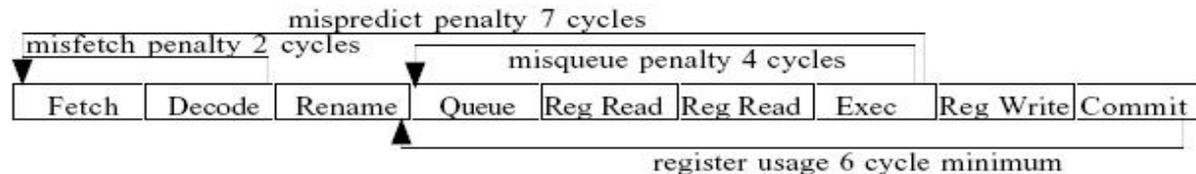
Exploiting choices... -3

- Register renaming
 - Much larger shared Reg. file (vs. previous scoreboarding)
 - Extra cycle for register access (in both RR and WB stages)
 - Increased misprediction penalty
 - More pipeline stage → more in-flight instructions → more strain on register file
- Optimistic issue
 - One-cycle increase of load latency; delayed scheduling □
 - Squashing load experiencing miss or bank conflict; re-issue

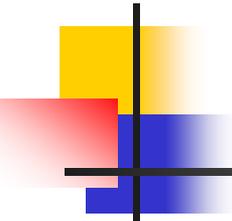


(a)

Pipeline of conventional superscalar (a) vs. modified for SMT (b) (from [Tullsen '96])



(b)

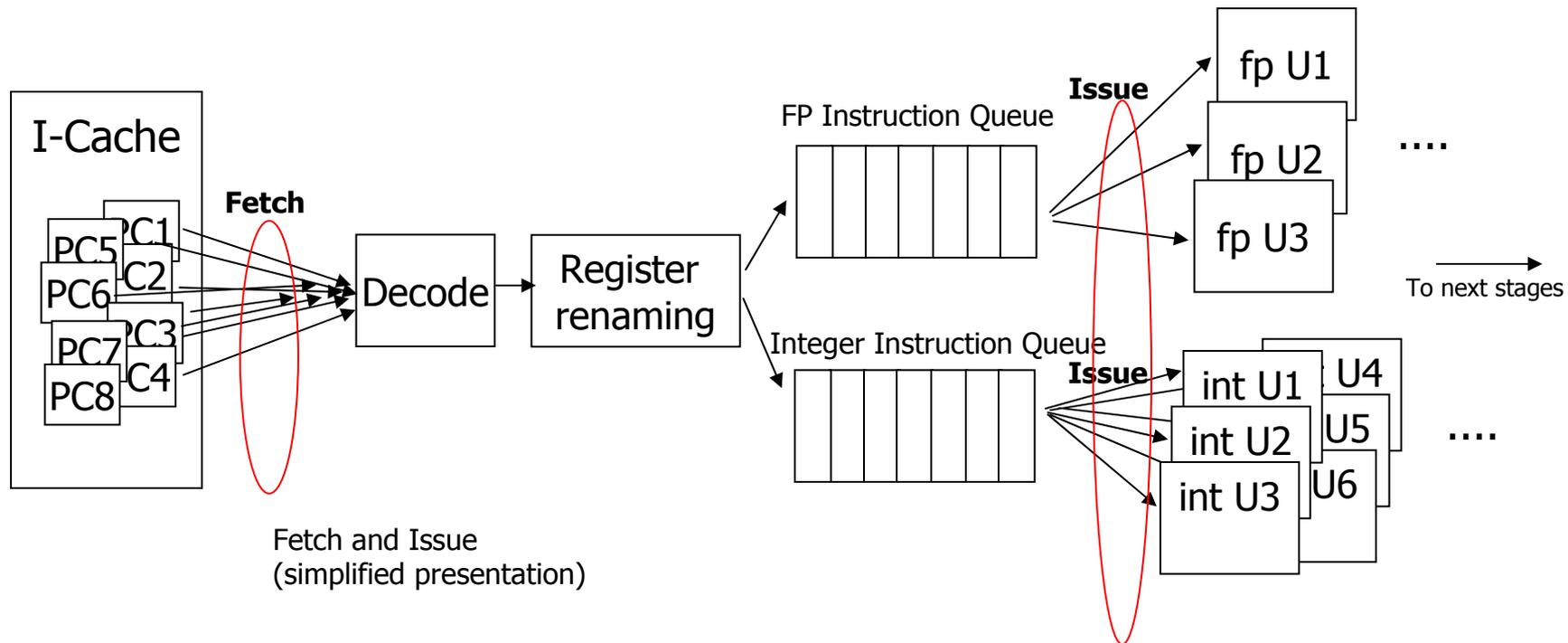


Exploiting choices... -4

- Base architecture of Alpha 21164
 - 9-issue wide; 3 fp, 6 int (4 can do ld/st); all pipelined
 - But with 8-instruction wide fetch & decode (limiting IPC) □
 - 32-entry instruction queues (int and fp); not FIFO □
 - Multi-banked (single port/bank) lock-up free caches; TLB
 - Not picking threads conflicting on an I-bank at fetch
 - Dynamic scheduling in HW; No static trace-scheduling
- Workload
 - SPEC92 benchmark suite
 - Multiprogramming workload of single-threaded applications (rather than parallel programs); more dynamic behavior

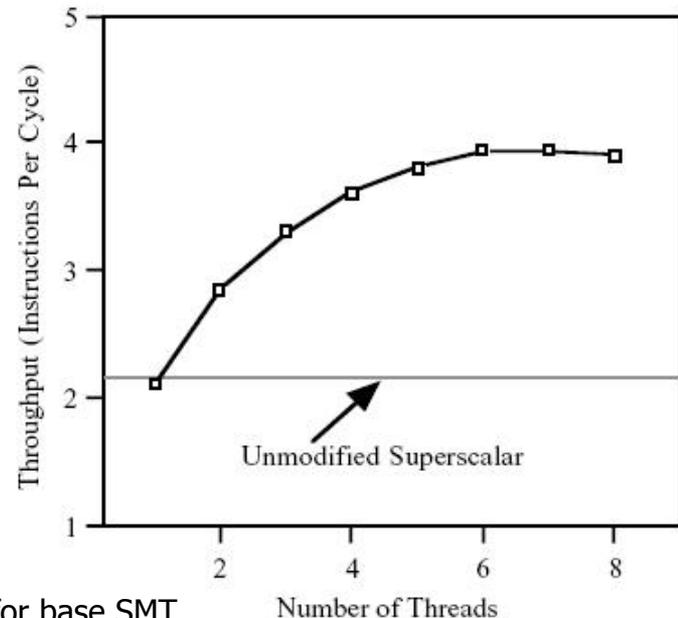
Exploiting choices... -5

- What this work focuses on: Fetch & Issue



Exploiting choices... -6

- Base architecture performance
 - Only 2% drop in single-threaded performance (compared to superscalar) 🛎 → due to longer pipeline
 - Peak throughput 84% higher than superscalar
 - Still not reaching 50% of issue-width
 - No FU type overloaded
 - Issue BW not bottleneck; proper instr. not reaching FUs
 - Out-of-register not often (3-7%)
 - Not significant mispred increase
 - Contention in IQ (full condition in 12-21% of cycles) *w/o FU contention*
 - high fetch throughput?? No!
 - Only 4.2 inst/cycle fetched
 - Guesses are:
 - Small IQ size
 - Low fetch throughput
 - Lack of parallelism (only 4/27)



Throughput for base SMT
(from [Tullsen '96])

Exploiting choices... -7

- 1) Improving fetch throughput by increasing fetch:
 - a) Efficiency: Fetching from multiple threads
 - b) Quality: Selecting better threads to fetch from (not round-robin)
 - c)- Availability: Reducing chance of blocking at fetch
<None possible on a superscalar processor when only one thread>
- a) Partitioning fetch BW
 - *Fetch block fragmentation:* (hard to fill 8 inst/cycle BW from *only one thread*)
 - a) frequent branches b) misalignment □
 - ➔ Spread the burden; but not too fine ➔ *thread shortage; more bank conflict*
 - Keep the total fetch BW fixed: 8 instr./cycle
 - **alg.num1.num2** : [algorithm].[#threads fetched from/cycle].[max #insts fetched/thread]
 - Multiplexing PCs to each bank; multiple output buses from each bank; multiplexed/replicated tag logic

Exploiting choices... -8

a) Partitioning fetch BW (cont.)

Methods:

- **RR.1.8** (baseline method)

- **RR.2.4:**

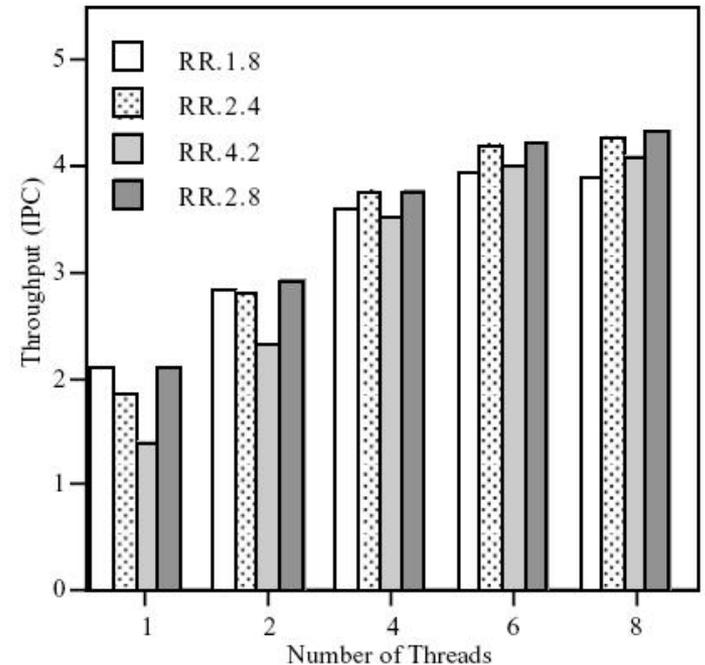
- 2 4-inst. output bus
- Good only with more than 2 threads □
- Poor single-thread

- **RR.4.2:**

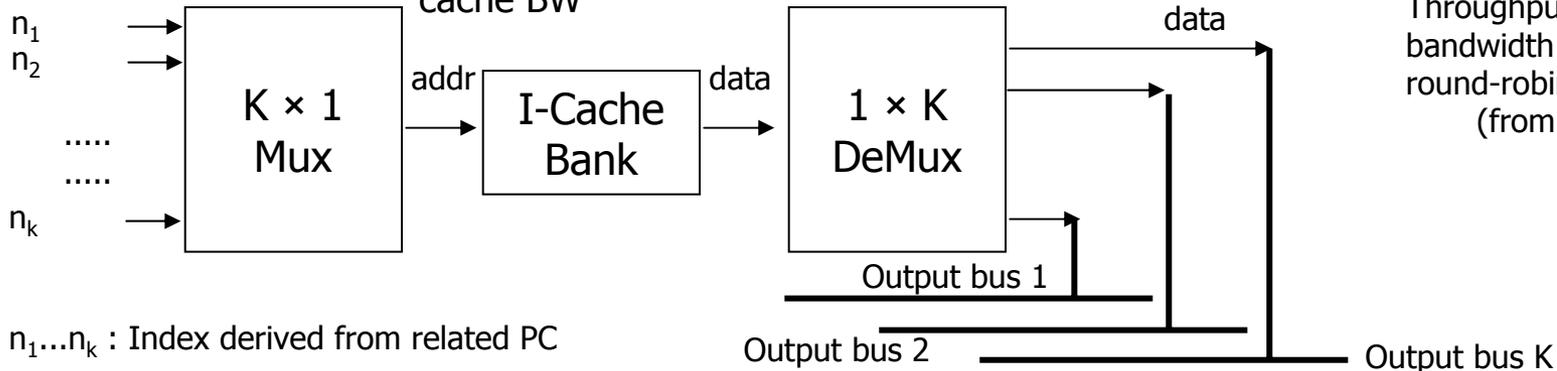
- 4 2-inst. output bus
- Suffers from thread shortage □

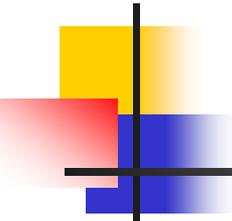
- **RR.2.8:**

- 2 8-instruction output bus (8 picked) □
- Best of both worlds: flexible partitioning
- 10% increase at 8 threads; Increase in I-cache BW



Throughput with different fetch bandwidth partitioning (still round-robin)
(from [Tullsen '96])





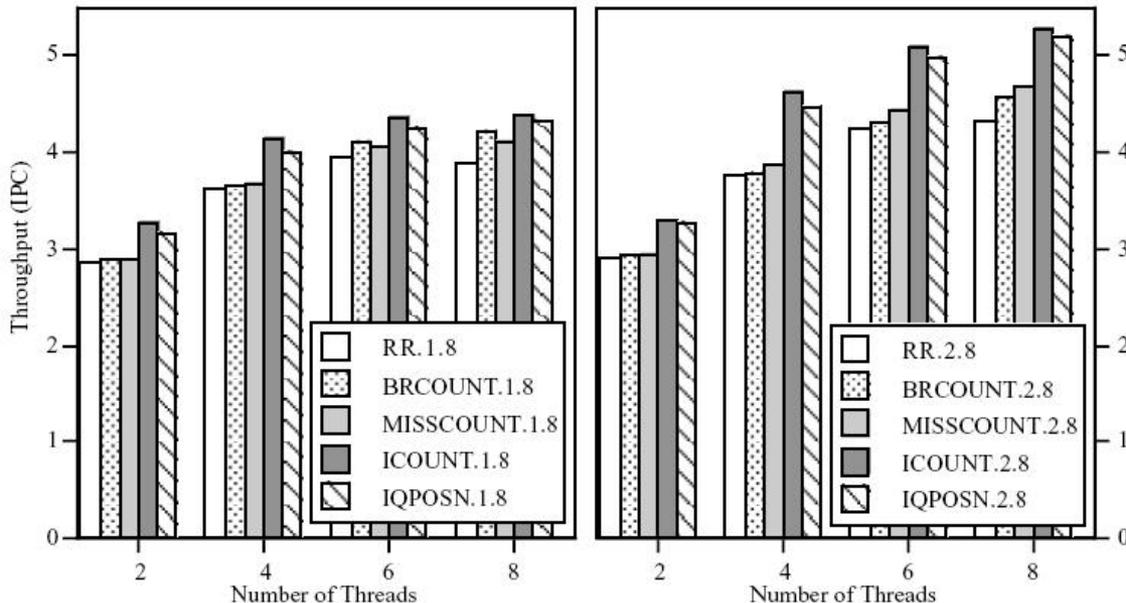
Exploiting choices... -9

- b) Selecting *good* threads
 - What are *bad* threads?
 - Probable to be in wrong path
 - Clogs IQ (stays in queue too long): D-cache miss; dependence chain
 - Delaying a bad thread rehabilitates it!
 - Its branch outcome will be known
 - Its dependency/latency gets resolved
 - Methods (each favors threads with fewest [X] in dec./ren/IQ stages):
 - BRCOUNT: X= unresolved branches
 - MISSCOUNT: X= D-cache misses
 - ICOUNT: X= instructions (any)
 - Prevents starvation (IQ be filled with one thread); better fairness/instr. mix
 - Higher priority to threads moving through IQ efficiently
 - IQPOSN: X= instructions close to IQ head (punishes old instr.!)
 - Easier to implement; doesn't need counter as previous three
 - Low feedback latency is important!
 - Instructions in queue/exec units fetched 3/6 cycles before!

Exploiting choices... -10

b) Selecting *good* threads (cont.)

- All better than round-robin
- ICOUNT is best (23% over RR.2.8); more issue-able inst. → reduced clog/full
- IQPOSN always within its 4%
- BRCCOUNT reduces wrong-path from 8.2 to 3.6% (throughput 8%)
 - But SMT already reduced wrong-path from 16 to 8% even with RR
- MISSCOUNT up to 8%; clog still high (12-14%)
- Intelligent fetching more important than partitioning (ICOUNT 1.8 vs. RR.2.8)



Metric	1 Thread	8 Threads	
		RR	ICOUNT
Int IQ full	7%	18%	6%
Fp IQ full	14%	8%	1%
Avg queue size	25	38	30
Out-of-register	3%	8%	5%

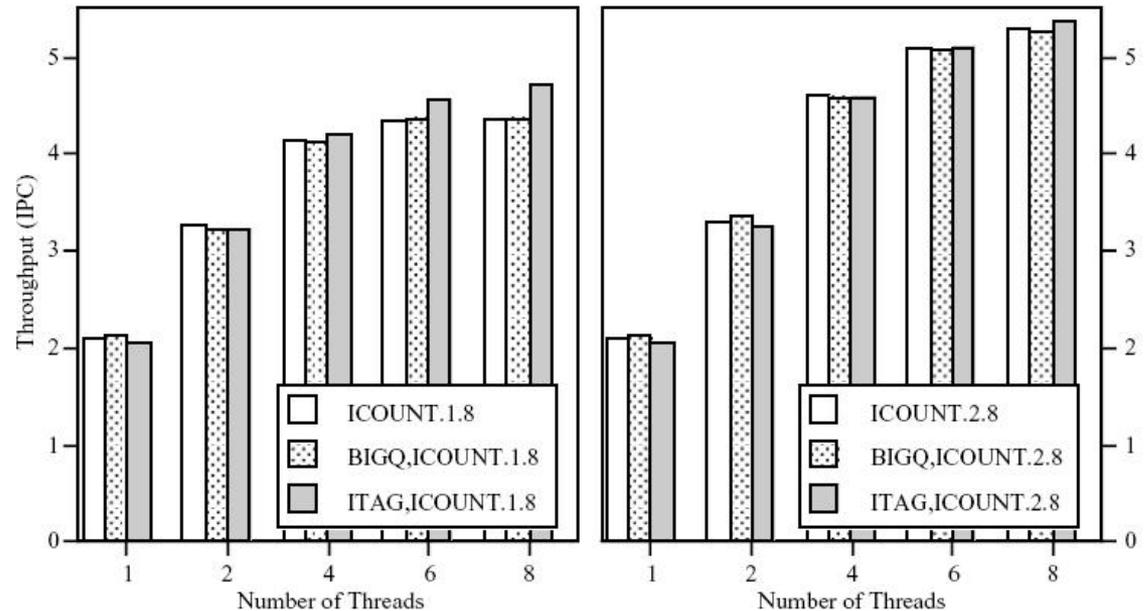
Throughput with different thread selection methods (from [Tullsen '96])

Exploiting choices... -11

c) Reducing probability of fetch blocking

- More efficient fetching → lose more when fetching blocks
- Causes: full IQ, I-cache miss
- Methods:
 - BIGQ: Doubling IQ size; inserting a non-searchable buffer HW
 - Not significant; out-of-date priority info; not as effective as ICOUNT
 - ITAG: Fetch sees a miss after attempt; let's do fetch a cycle earlier
 - Higher mispred penalty; more ports to I-tag to try replacements (e.g. 2→3 or 4)
 - Trade-off: 1.8 benefits more; cost of a missed fetch slot is higher; lower throughput for low number of threads b/c thread shortage, mispred penalty
- ITAG helps 8 threads 5.3→5.4 (37% over base SMT)

Throughput with ITAG, BIGQ
(from [Tullsen '96])



Exploiting choices... -12

- 2) Improving issue throughput:
 - Reducing # of issued useless instr. (wrong-path/optimistic)
 - Methods
 - OLDEST_FIRST: Default; priority to any instr. closer to IQ head
 - OPT_LAST: Optimistic instr. as late as possible
 - Minor decrease
 - SPEC_LAST: Speculative instr. as late as possible
 - No significant change
 - BRANCH_FIRST: Branch instruction as early as possible
 - Worse! Branches often load dependent → still optimistic issue
 - Sticking to default OLDEST_FIRST

Issue Method	Number of Threads					Useless instructions	
	1	2	4	6	8	wrong-path	optimistic
OLDEST_FIRST	2.10	3.30	4.62	5.09	5.29	4%	3%
OPT_LAST	2.07	3.30	4.59	5.09	5.29	4%	2%
SPEC_LAST	2.10	3.31	4.59	5.09	5.29	4%	3%
BRANCH_FIRST	2.07	3.29	4.58	5.08	5.28	4%	6%

Exploiting choices... -13

Bottlenecks: what to improve/relax

- Issue BW:
 - No. Infinite FUs increase throughput 0.5%
- IQ size:
 - Not after ICOUNT; Only 1% with 64-entry searchable queue
- Fetch BW:
 - 8% increase (5.7 IPC) if BW increased to 16 (8 from to 2 threads)
 - More pressure on IQ and Reg. file: 6.1 IPC with 140 regs, 64-entry
- Branch prediction:
 - Miss-rate increases (with higher threads); but SMT more miss-tolerant
 - Doubling BHT size, 2% increase
 - Perfect prediction: 25% 1 thread, 15% at 4 thread, 9% at 8 thread
- Memory/cache:
 - SMT is good at latency hiding; still very important
 - No bank conflict -> 3% increase
- Register file:
 - Infinite excess register (from 100) → 2%
 - ICOUNT reduces clog/strain on Reg. file

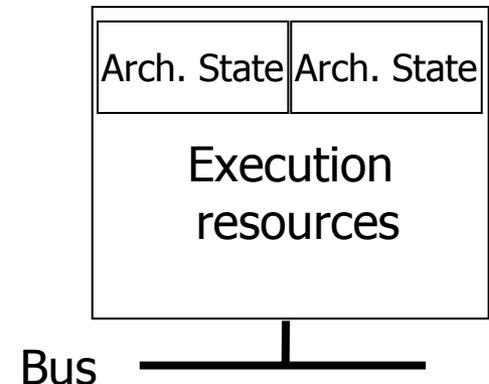
Instructions pass through queue faster; less in-flight instructions 30

- Summary

- Achieves the three goals:
 - High throughput gain (5.4 IPC; 2.5 fold increase over SS)
 - w/o extensive change to conventional superscalar
 - w/o compromising single-threaded performance
- Through advantageous of SMT having multiple threads (TLP); not possible on superscalar
- Mainly from:
 - Fetch BW *flexible partitioning*
 - Being *selective* about thread to fetch from → *Intelligent fetch*

Pentium4

- Intel calls it Hyperthreading
 - Started with Xeon; targeting servers
 - Call it “Significant new technology direction for Intel”
 - Adds less than 5% to chip area and power requirement
 - Performance increase of 16-28% on server-centric benchmarks
 - Design goals:
 - Minimize die-area cost (cost containment/conservative approach)
 - Prevent a stalled thread blocking other
 - > proper management of buffering queues
 - Not degrading single-threaded scenario
 - > partition/duplicate buffering queues
 - Architectural state (context):
 - General-purpose registers
 - APIC (Advanced Programmable Interrupt Controller)
 - Control & state registers

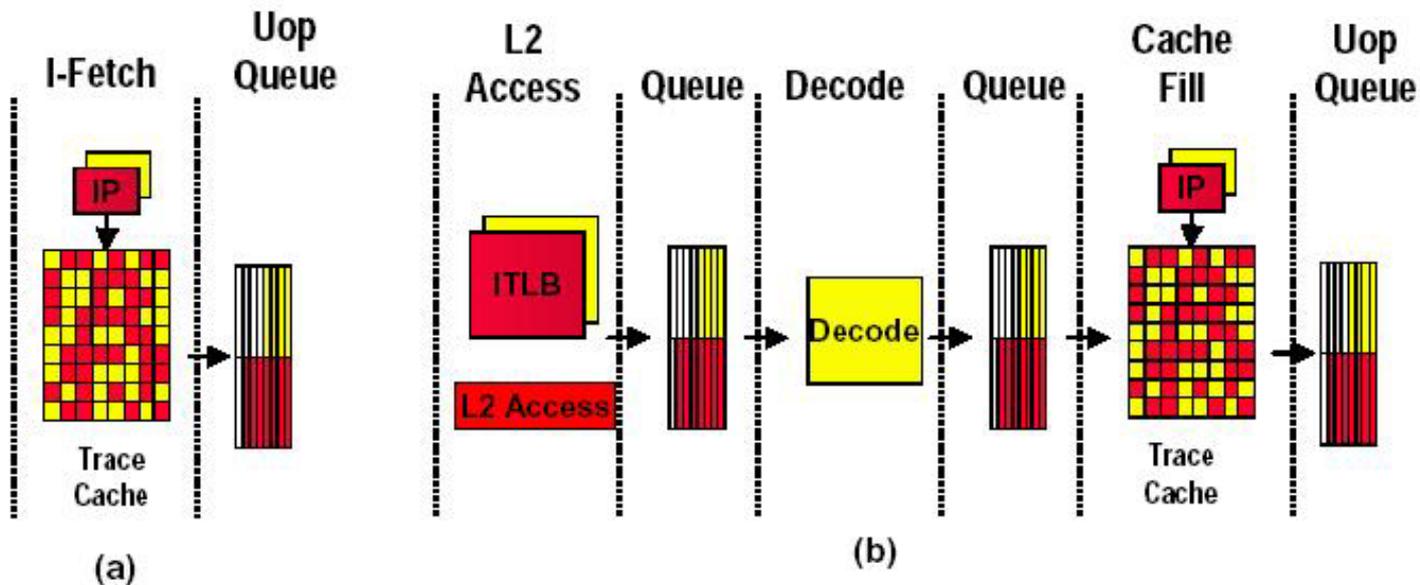


- Challenges:
 - New algorithms to share resources; preventing deadlock/livelock
 - High logic complexity for already complex IA-32 architecture
 - Whole new validation complexity
 - Many combinations, events, interactions
 - More complex interactions than multiprocessor validation of SMP
- Sharing policy:
 - Partition:
 - For highly utilized resources but in unpredictable rate; e.g. major pipeline queues
 - Preventing an slow thread piling up queues blocking a fast thread
 - Threshold:
 - For bursty used resources like schedulers
 - Full sharing:
 - For large structures; e.g. caches

- Front-end:

- Trace Cache (TC)

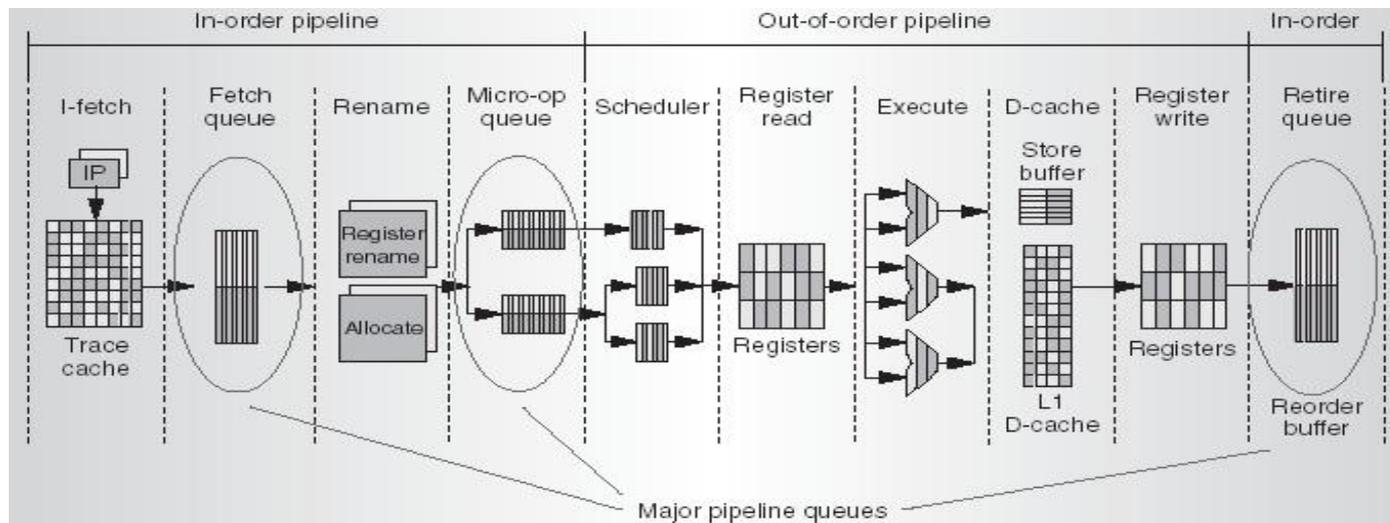
- Keeps decoded instructions (uops) for both
 - Shared; entries tagged with thread ID
 - Two instruction pointers
 - Round-robin access (if both threads want to fetch at same cycle)
 - Two microcode inst. pointers for Microcode ROM



Trace cache:
 a) Hit b) Miss
 (from [Marr'02])

- Front-end (cont.):
 - ITLB:
 - Separate
 - L2 cache interface:
 - Shared; First come/first serve; at least one request/thread is guaranteed
 - Decoding logic:
 - Shared; alternates between threads
 - Return stack buffer:
 - Duplicated; small
 - Branch prediction:
 - BHT array is shared; but tagged with thread ID
 - Fetch/Uop Queue:
 - Duplicated; decouples front-end from execution engine

- Back-end:
 - Allocator: Takes up from queue and allocates several buffers
 - Partitioned; Total(reorder:126, load:48, store:24)
 - Shared; Physical registers:128 int;128 fp
 - Thread stalls if all its buffer used up
 - Register renaming:
 - Separate Register Alias Table (RAT)
 - Instruction queues (partitioned):
 - Load/Store queue
 - General instruction queue



Netburst u-arch.
execution pipeline
(from [Koufaty '03])

- Back-end: (cont.)
 - Scheduler queues:
 - Shared; receive uops from instruction queues in alternating cycles
 - Threshold on max. # of uops entering from a thread -> avoid starvation
 - Dispatch is regardless of thread; only operand availability
 - Small; runs at high speed (twice the clock freq; e.g. 6GHz)
 - Execution units: Thread unaware; simple forwarding logic
 - Retirement:
 - Partitioned reorder buffers
 - Alternates between threads every cycle
 - DTLB: Shared
 - Caches:
 - All levels shared; potential for *both* conflict and data-sharing (prefetching by helper thread); high associativity degree used
- Modes of operation (HALT instr):
 - Single-task (ST): Partitioned resources are recombined (ST0/ST1) □
 - Multi-task (MT): Both architectural resources are active

Future & notes

■ Future

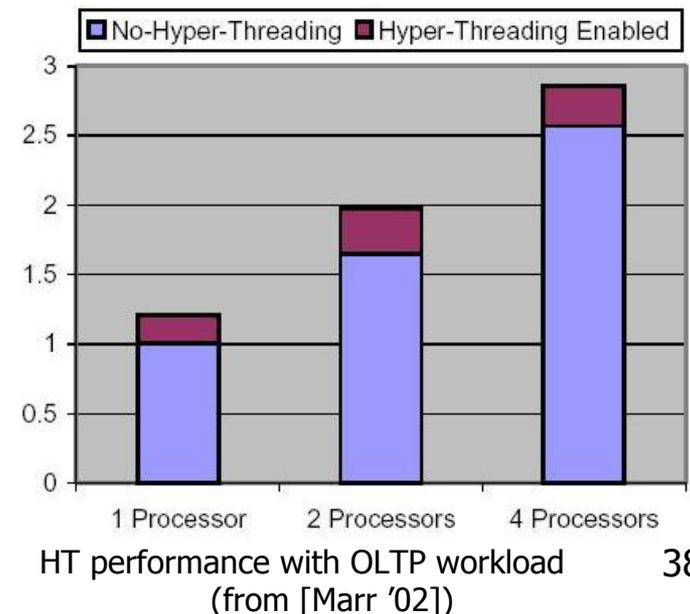
- Cache prefetching/pre-computation/pre-execution:
 - Identify delinquent loads
 - Create pre-computation slices (p-slice)
 - Execute them in another thread parallel to main thread
 - SW based: Next week's presentation
 - HW based: Hardware-constructed thread-based cache prefetcher
 - Instruction analysis, extraction and optimization through hardware; 14-33% increase for memory-intensive benchmarks.

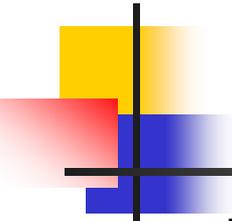
■ Better multithreaded application development tools

- OpenMP compiler

■ Software notes:

- When multiple threads, each runs slower than if was running exclusively (naturally)!
- Comparison against MP →→
- On MP systems with SMT processors, OS should schedule software threads to different processors first; Linux 2.4 kernel problem





Main References

- [Orlowski '01] Andrew Orlowski, "Project Jackson - Why SMT is the joker in the chip pack," *The Register*, Feb. 2001, <http://www.theregister.co.uk/content/2/17165.html>
- [Tullsen '95] D.M. Tullsen, S.J. Eggers, and H.M. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," In proceedings of the *22nd Annual International Symposium on Computer Architecture*, June 1995, pages 392-403.
- [Tullsen '96] D.M. Tullsen, S.J. Eggers, J.S.Elmer, H.M. Levy, R. Stamm, "Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor," In proceedings of the *23rd Annual International Symposium on Computer Architecture*, May 1996.
- [Koufaty '03] David *Koufaty*, *Deborah Marr*, "Hyperthreading technology in the netburst microarchitecture," *IEEE Micro*, Vol. 23, Issue: 2 , March-April 2003, pages:56-65.
- [Marr '02] Deborah T. Marr, et al., "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology Journal*, Vol. 06, Issue 01, Feb. 2002, pages 4-15.
- [Shen '02] John Paul Shen, "From Instruction Level To Thread Level Parallelism," *Microprocessor Research Forum*, October 29, 2002.