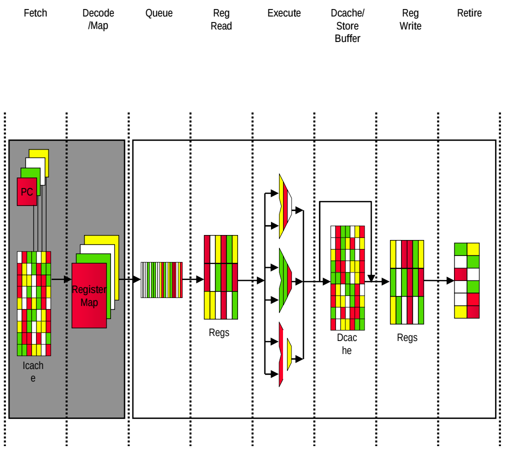
18-742
Lecture 25

Multithreading

Spring 2005
Prof. Babak Falsafi
http://www.ece.cmu.edu/~ece742



Slides developed in part by Prof. Falsafi from Mukherjee from Intel.

What & Why of Multithreading

What?

- Run multiple threads on the same core

Why?

- Key hardware resources are often idle for a single thread
- Large variability in IPC across threads
- E.g., desktop 2-4 IPC for 8-way, OLTP 0.4 IPC for 8-way

How?

- Multiple thread contexts
  - » Architectural: PC, SP, reg file
  - » Microarchitectural: Fetch, LSQ, ROB

Core Sharing

Time sharing:

- Run one thread
- On a long-latency operation (e.g., cache miss), switch
- Also known as “switch-on-miss” multithreading

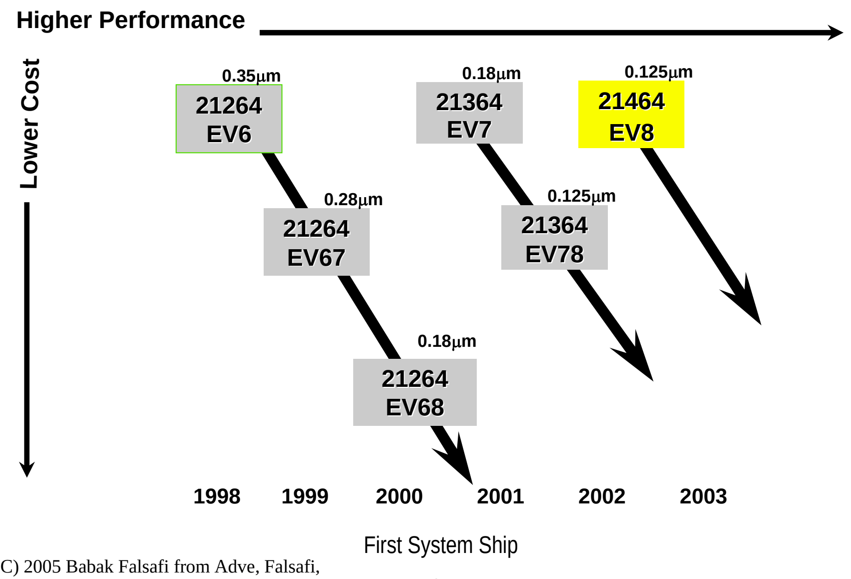
Space sharing:

- Across pipeline depth
  - Fetch and issue each cycle form a different thread
- Both across pipeline width and depth
  - Fetch and issue each cycle from from multiple threads
  - Policy to decide which to fetch gets complicated
  - Also known as “simultaneous” multithreading
  - E.g., Alpha 21464, IBM POWER5

Alpha 21264 Microprocessor

- Architectural Features
  - First “Out-of-Order” Alpha
  - Four-wide superscalar
  - …
- Performance
  - World’s Fastest Microprocessor (www.spec.org, 11/17/99)
  - 39 SPECINT95, 68 SPECFP95 @ 700 Mhz
    - Intel Pentium III @ 733 Mhz delivers 36 SPECINT95, 30 SPECFP95

Alpha Microprocessor Overview


Higher Performance

Lower Cost

0.35µm 21264 EV6

0.28µm 21264 EV67

0.18µm 21264 EV68

0.18µm 21364 EV7

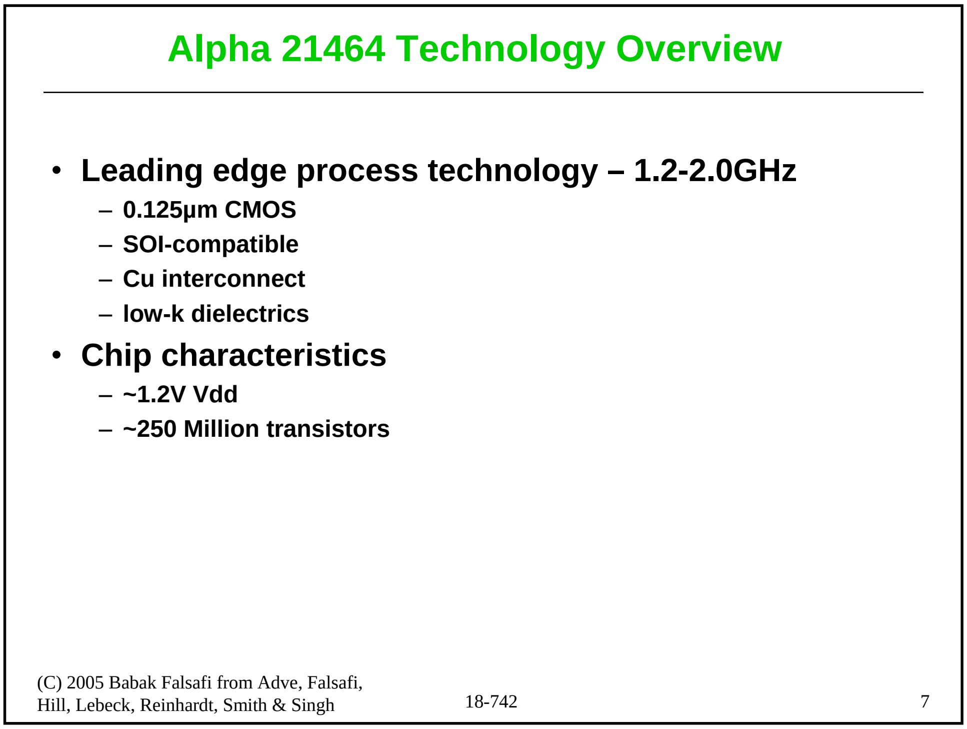
0.125µm 21364 EV78

0.125µm 21464 EV8

First System Ship

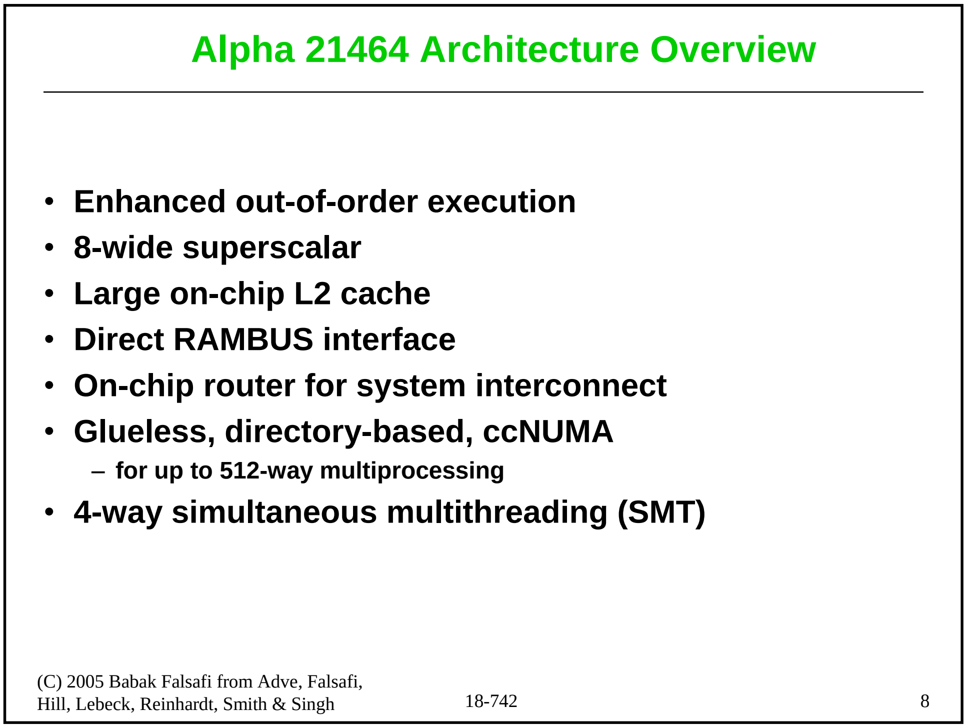
Alpha 21464 Goals

- **Leadership single thread performance**
  - **Higher operating frequency / better technology**
  - **New microarchitecture**
  - **Integrated memory interface (like 21364)**
- **Leadership multiprocessor performance**
  - **Simultaneous Multithreading (with minimal change/cost)**
  - **Integrated system / multiprocessor interface (like 21364)**



Alpha 21464 Technology Overview

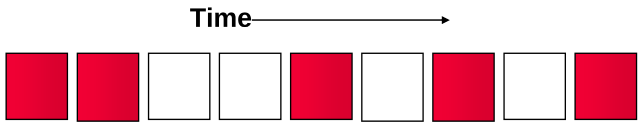
- Leading edge process technology – 1.2-2.0GHz
  - 0.125µm CMOS
  - SOI-compatible
  - Cu interconnect
  - low-k dielectrics
- Chip characteristics
  - ~1.2V Vdd
  - ~250 Million transistors



Alpha 21464 Architecture Overview

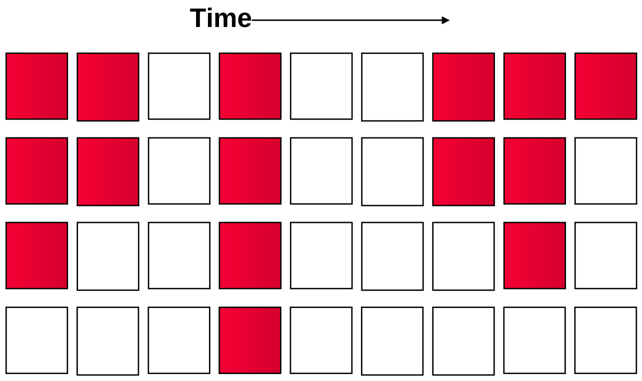
- Enhanced out-of-order execution
- 8-wide superscalar
- Large on-chip L2 cache
- Direct RAMBUS interface
- On-chip router for system interconnect
- Glueless, directory-based, ccNUMA
  - for up to 512-way multiprocessing
- 4-way simultaneous multithreading (SMT)

Instruction Issue



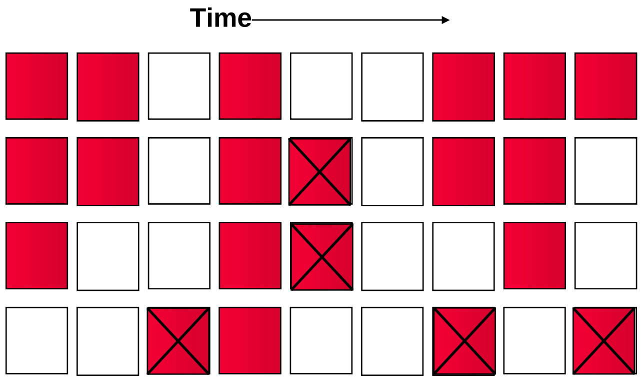
Reduced function unit utilization due to dependencies

Superscalar Issue



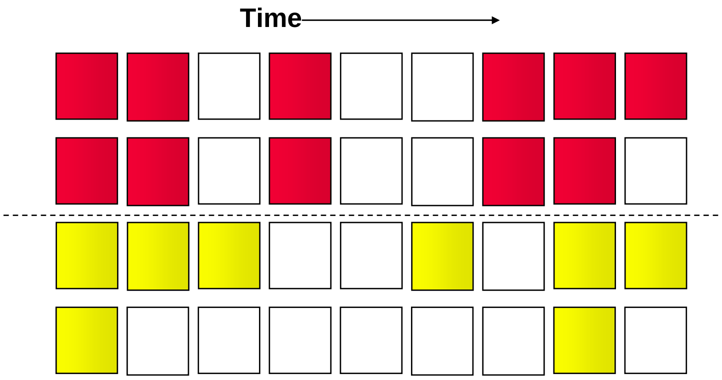
Superscalar leads to more performance, but lower utilization

Predicated Issue

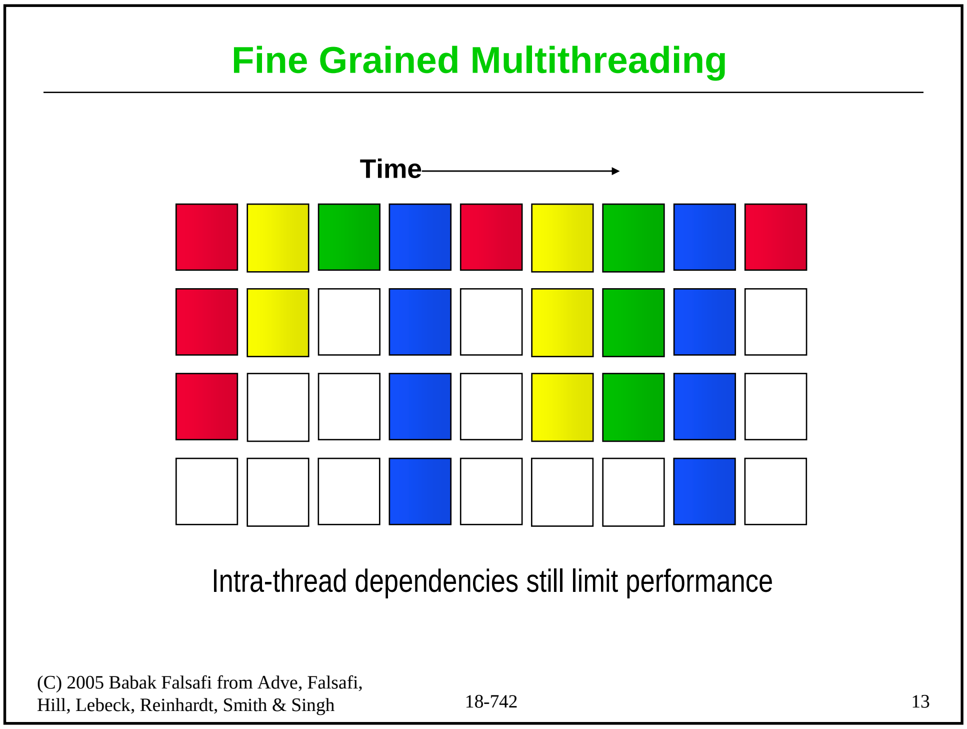


Adds to function unit utilization, but results are thrown away

Chip Multiprocessor

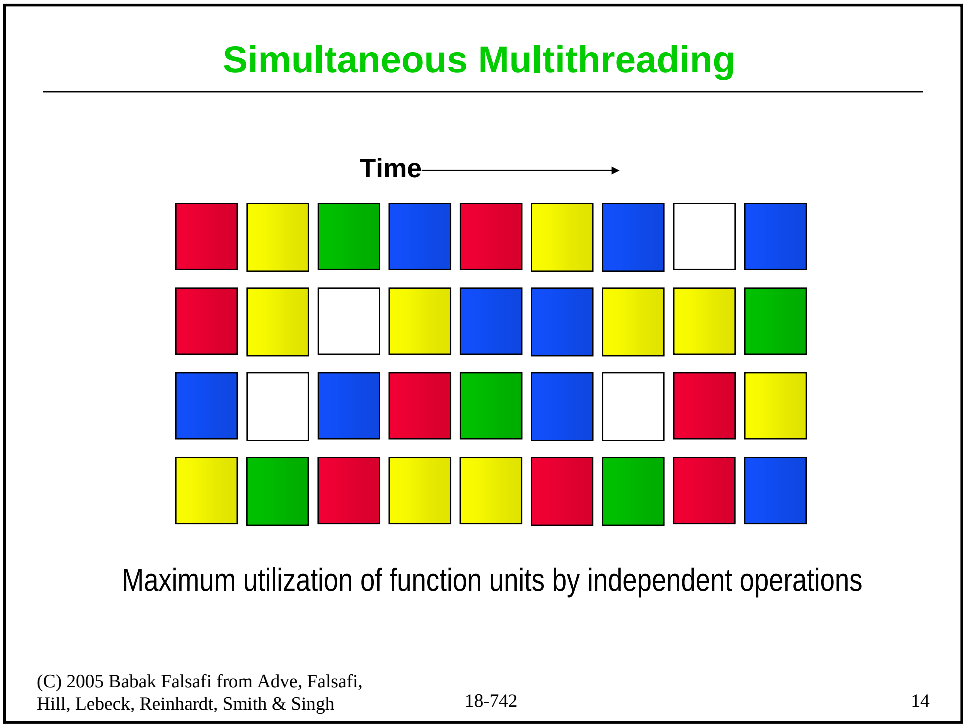


Limited utilization when only running one thread



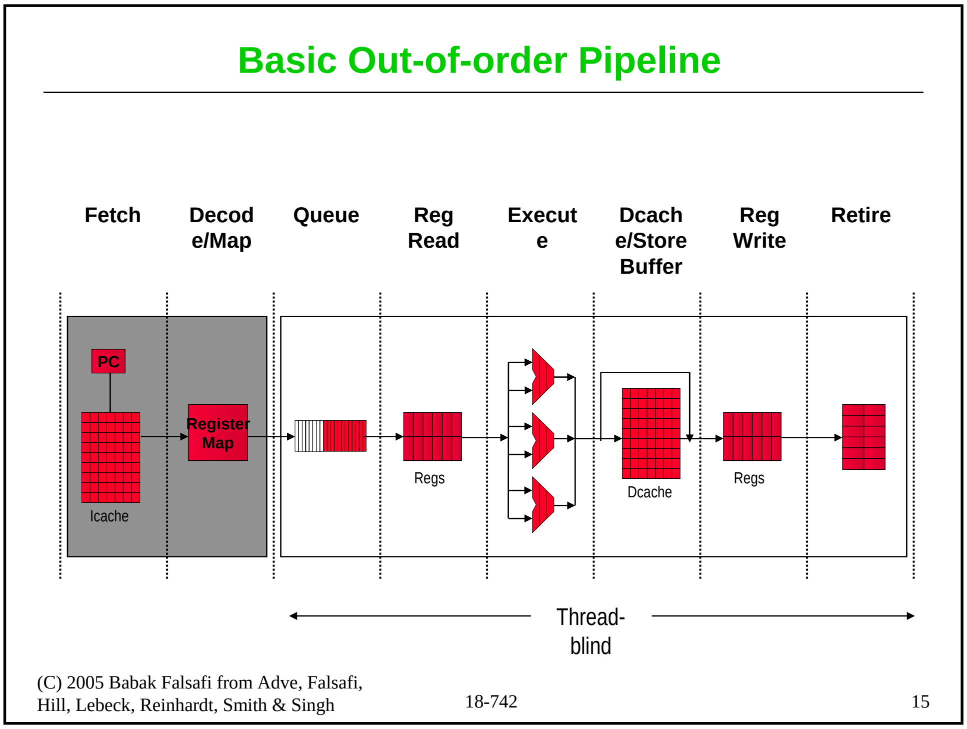
Fine Grained Multithreading

Intra-thread dependencies still limit performance



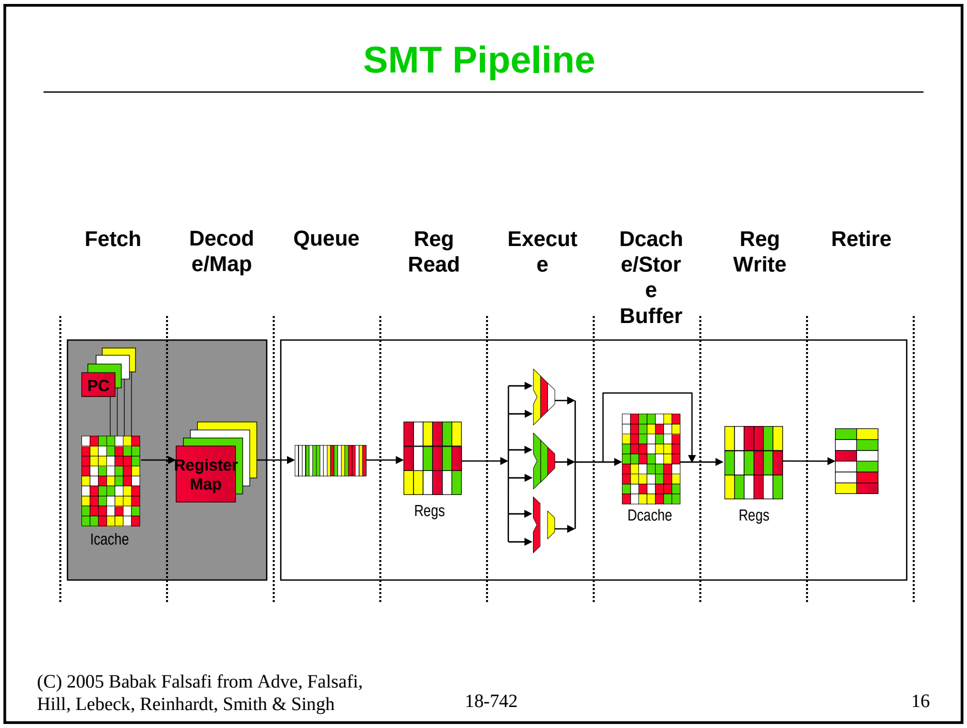
Simultaneous Multithreading

Maximum utilization of function units by independent operations



Basic Out-of-order Pipeline

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



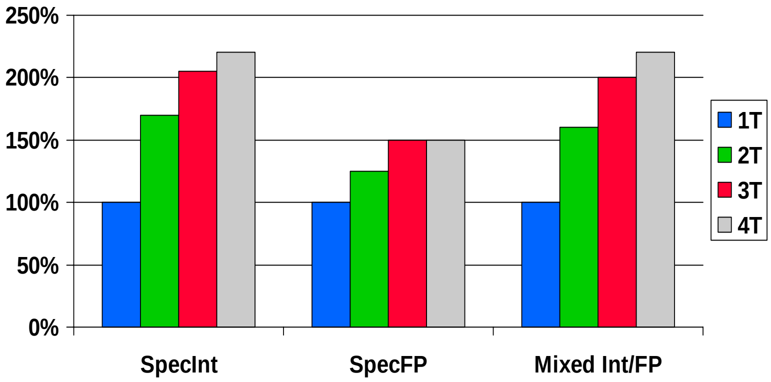
SMT Pipeline

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

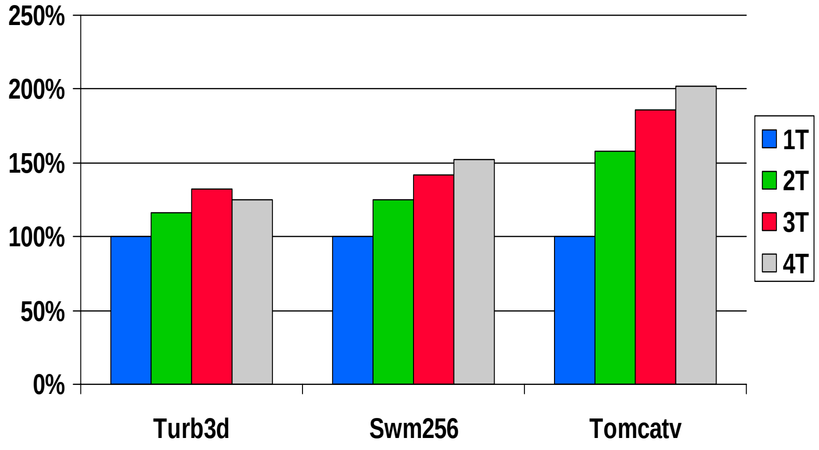
Changes for SMT

- Basic pipeline – unchanged
- Replicated resources
  - Program counters
  - Register maps
- Shared resources
  - Register file (size increased)
  - Instruction queue
  - First and second level caches
  - Translation buffers
  - Branch predictor

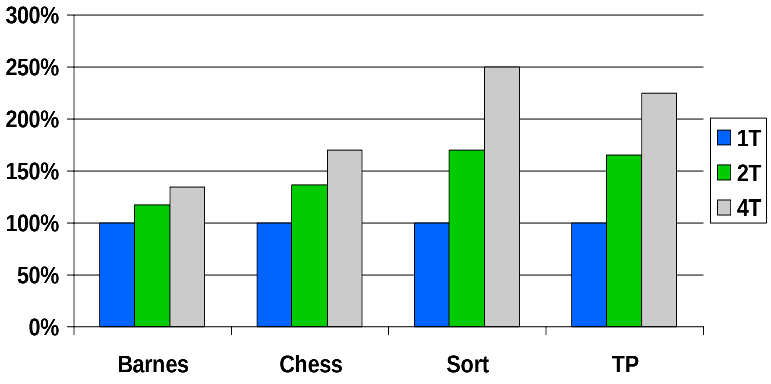
Multiprogrammed workload

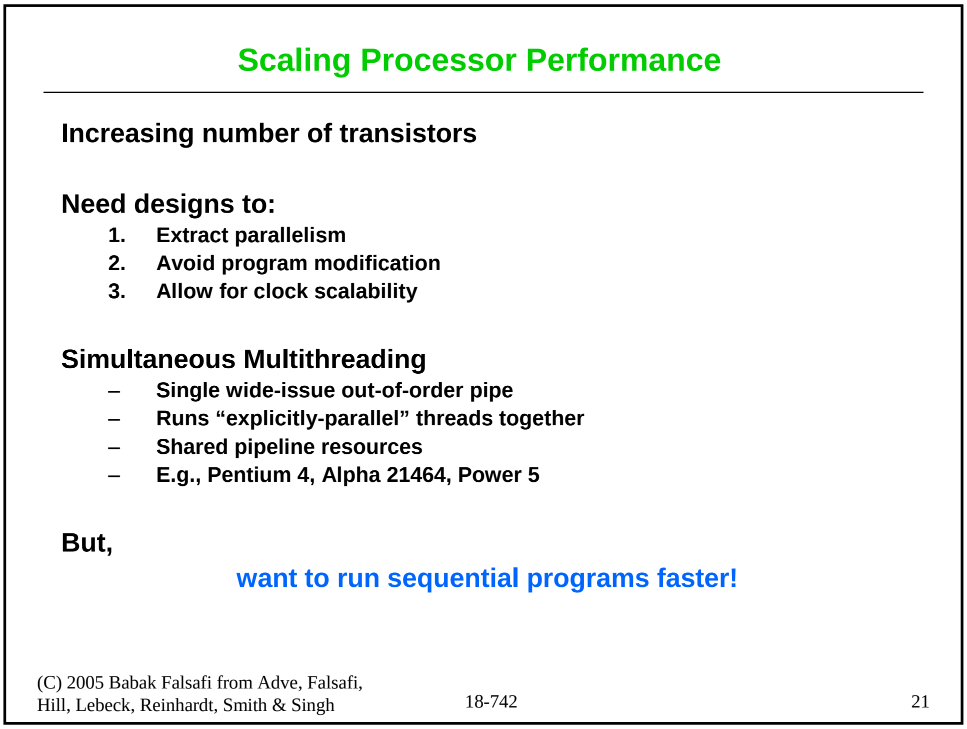


Decomposed SPEC95 Applications



Multithreaded Applications





Scaling Processor Performance

Increasing number of transistors

Need designs to:

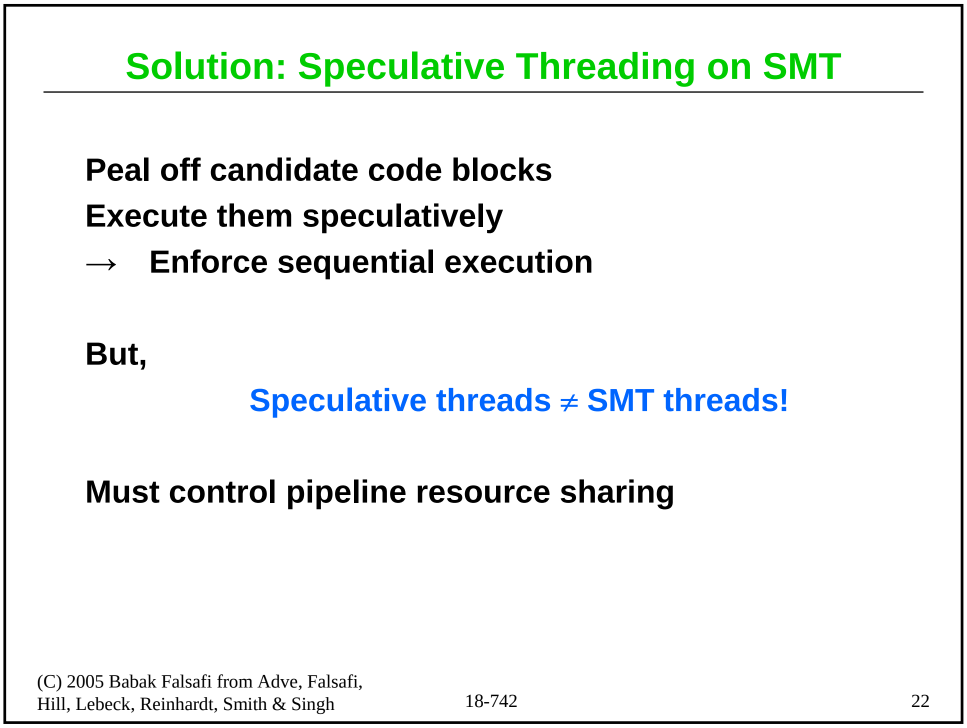
1. Extract parallelism
2. Avoid program modification
3. Allow for clock scalability

Simultaneous Multithreading

- Single wide-issue out-of-order pipe
- Runs “explicitly-parallel” threads together
- Shared pipeline resources
- E.g., Pentium 4, Alpha 21464, Power 5

But,

want to run sequential programs faster!



Solution: Speculative Threading on SMT

Peal off candidate code blocks

Execute them speculatively

→ Enforce sequential execution

But,

Speculative threads ≠ SMT threads!

Must control pipeline resource sharing

Implicitly-Multithreaded processors [Park et al., ISCA’03]

- Speculative threading on SMT
- Compiler-specified threads (Multiscalar)
- Hardware speculation + verification
- 3 uArch optimizations to throttle sharing

Speedup over superscalar by 20%-29%

IMT Architecture

Compiler

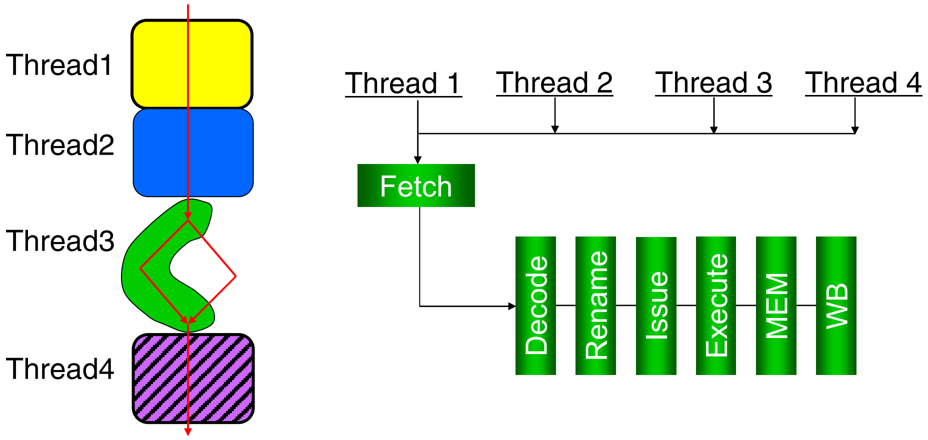
- Selects candidate threads
- Provides register + control information

Hardware

- Control Dependence
- Register communication
- Memory disambiguation

Compiler + Hardware integration

Speculative Threading on SMT



IMT Hardware

Control Dependence

- **Uses compiler information**
- **Minor change on branch prediction**

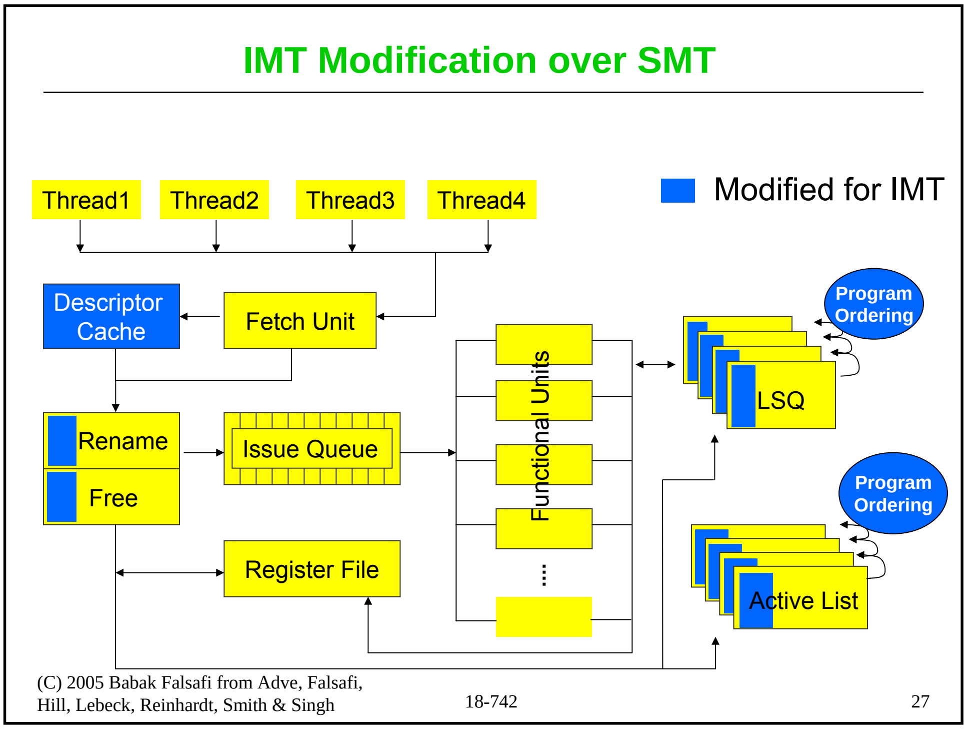
Register communication

- **Uses compiler information**
- **Leverages conventional renaming**

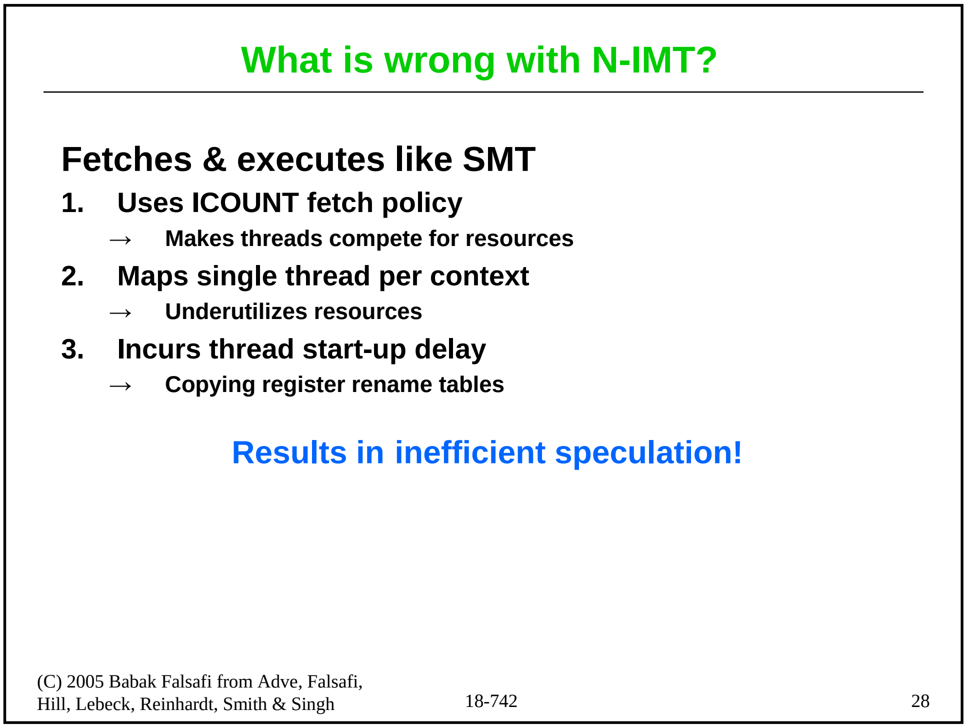
Memory disambiguation

- **Searches across ld/st queues**

Naïve-IMT (N-IMT) → Minor mods to SMT



IMT Modification over SMT



What is wrong with N-IMT?

Fetches & executes like SMT

1. **Uses ICOUNT fetch policy**
   - → **Makes threads compete for resources**
2. **Maps single thread per context**
   - → **Underutilizes resources**
3. **Incurs thread start-up delay**
   - → **Copying register rename tables**

Results in inefficient speculation!

Efficient Speculation: Optimized IMT

O-IMT uses 3 uArch mechanisms:

1. **Resource- & Dep.-based fetch**
   - → **Parallel fetch only from independent threads**
2. **Context multiplexing**
   - → **Map multiple threads per context**
3. **Hide thread start-up overhead**

   (covered in the paper)

Opt(1) Res.-Based Fetch: Example

Free Registers: 0

N-IMT

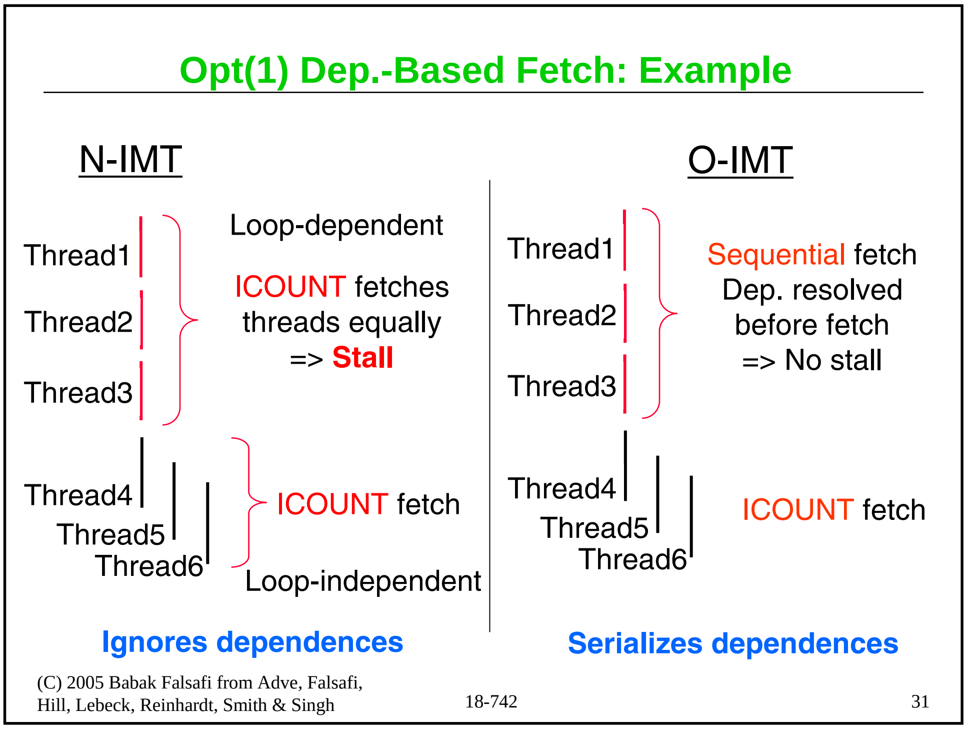
	Occupied Registers
Thread1	5 +10 +20
Thread2	5 +10
~~Thread3~~	~~5 +10~~
~~Thread4~~	~~5 +10~~

Naïve resource allocation

O-IMT

	Occupied Registers	**Prealloc. Registers**
Thread1	5 +10 +20	**35**
Thread2	5 +10	**15**
Thread3	5 +5	**10**
Thread4		

Careful resource allocation



Opt(1) Dep.-Based Fetch: Example

N-IMT

Thread1

Thread2

Thread3

Loop-dependent

ICOUNT fetches threads equally => **Stall**

Thread4

Thread5

Thread6

ICOUNT fetch

Loop-independent

Ignores dependences

O-IMT

Thread1

Thread2

Thread3

Sequential fetch
Dep. resolved
before fetch
=> No stall

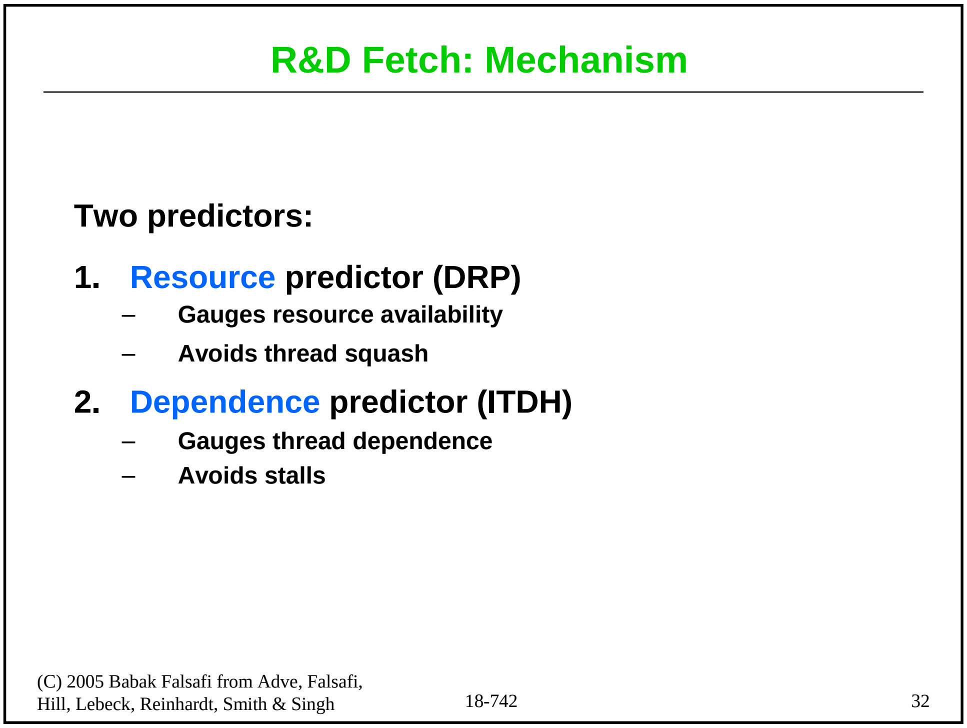
Thread4

Thread5

Thread6

ICOUNT fetch

Serializes dependences



R&D Fetch: Mechanism

Two predictors:

1. **Resource predictor (DRP)**
   - **Gauges resource availability**
   - **Avoids thread squash**
2. **Dependence predictor (ITDH)**
   - **Gauges thread dependence**
   - **Avoids stalls**

Opt(2): Context Multiplexing

Insufficient instruction overlap because

- – Maps one thread to a context
- – A few contexts (2-8) in SMT
- – Small Multiscalar threads (15-20 inst/thread)
- – Variable thread size (up to 100 insts)

But, [Hammond et al., ASPLOS98] [Vijaykumar et al., Micro98]

Larger threads → higher squash overhead

Context Multiplexing: Mechanism

Splits context resources dynamically

Assigns multiple threads/context

As many as allowed by resources:

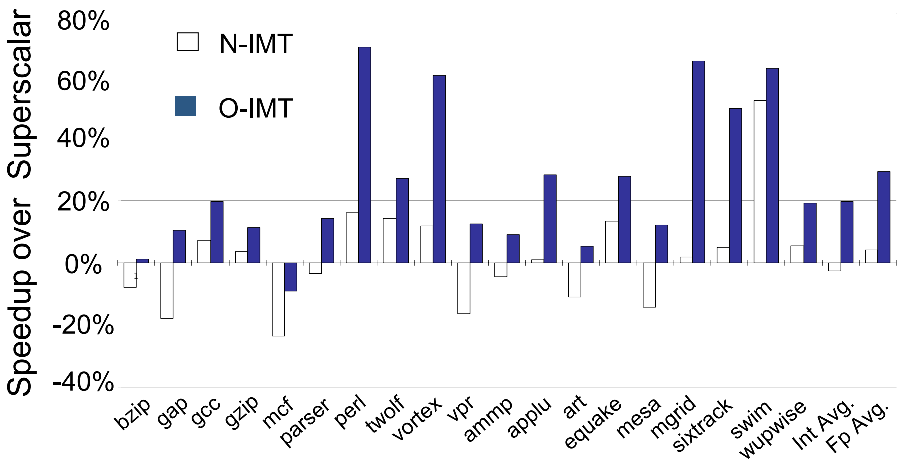
- – Active list entries
- – Ld/St queue entries

Increases effective thread size dynamically!

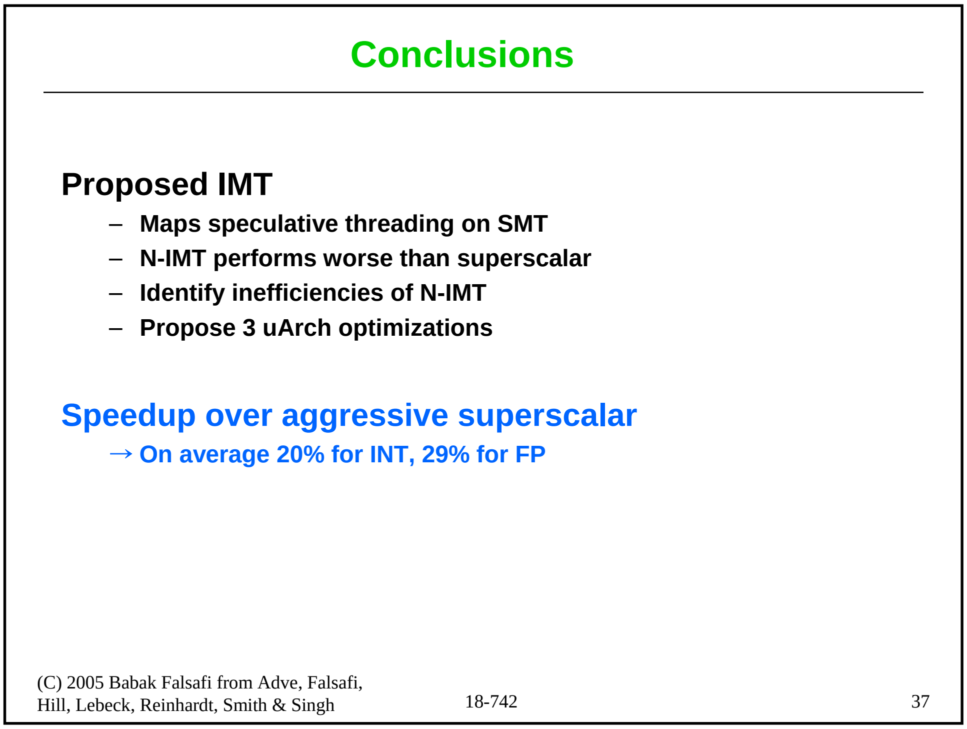
Methodology: Detailed Simulation

<table>
<tr><th>IMT</th><th>Superscalar</th></tr>
<tr><td>8 contexts
128-entry active list,
32-entry LSQ per context</td><td>1 context
1024-entry active list,
256-entry LSQ</td></tr>
<tr><th colspan="2">IMT &amp; Superscalar</th></tr>
<tr><td colspan="2">356-entry register file INT/FP
two fetch ports (multiple predictions for superscalar)
8-way issue
64-entry Issue queue</td></tr>
</table>

Base System Comparison



O-IMT better than N-IMT by 24%



Conclusions

Proposed IMT

- Maps speculative threading on SMT
- N-IMT performs worse than superscalar
- Identify inefficiencies of N-IMT
- Propose 3 uArch optimizations

Speedup over aggressive superscalar

→ On average 20% for INT, 29% for FP