New Paradigms for Instruction Level Parallelism

J. E. Smith
Univ. of Wisconsin-Madison

October 1995

Defn: Instruction Level Parallelism

- Very fine grain

  Single instructions or a few instructions

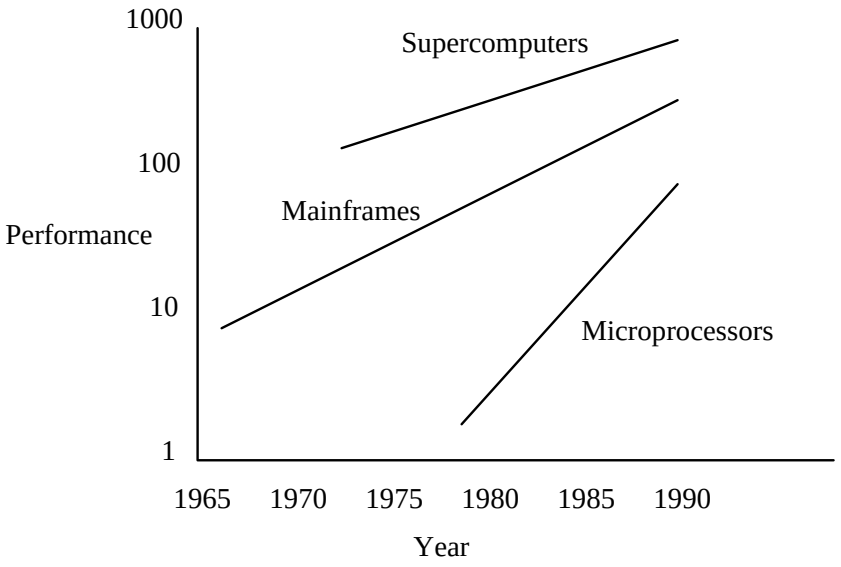
- Compiler-detected

- Existing instruction sets?

  Not a requirement, but an important practical consideration

ILP Progress to Date

- Typical graph comparing micros to mainframes:



- Micros look impressive, but they have borrowed heavily from mainframes & supers (or re-invented)

  RISC, pipelining, dynamic inst. issue, caches, branch prediction, pipelined memory paths, compilation techniques etc.

- 30-40 years of progress have been compressed into 15-20 years of micro history

- What next? ...Not much left to borrow

Current ILP Paradigms

- Superscalar

  the incumbent

- VLIW

  getting lots of attention in some circles

  unproven for non-numeric codes
  (arguably proven for numeric codes)

- Vectors

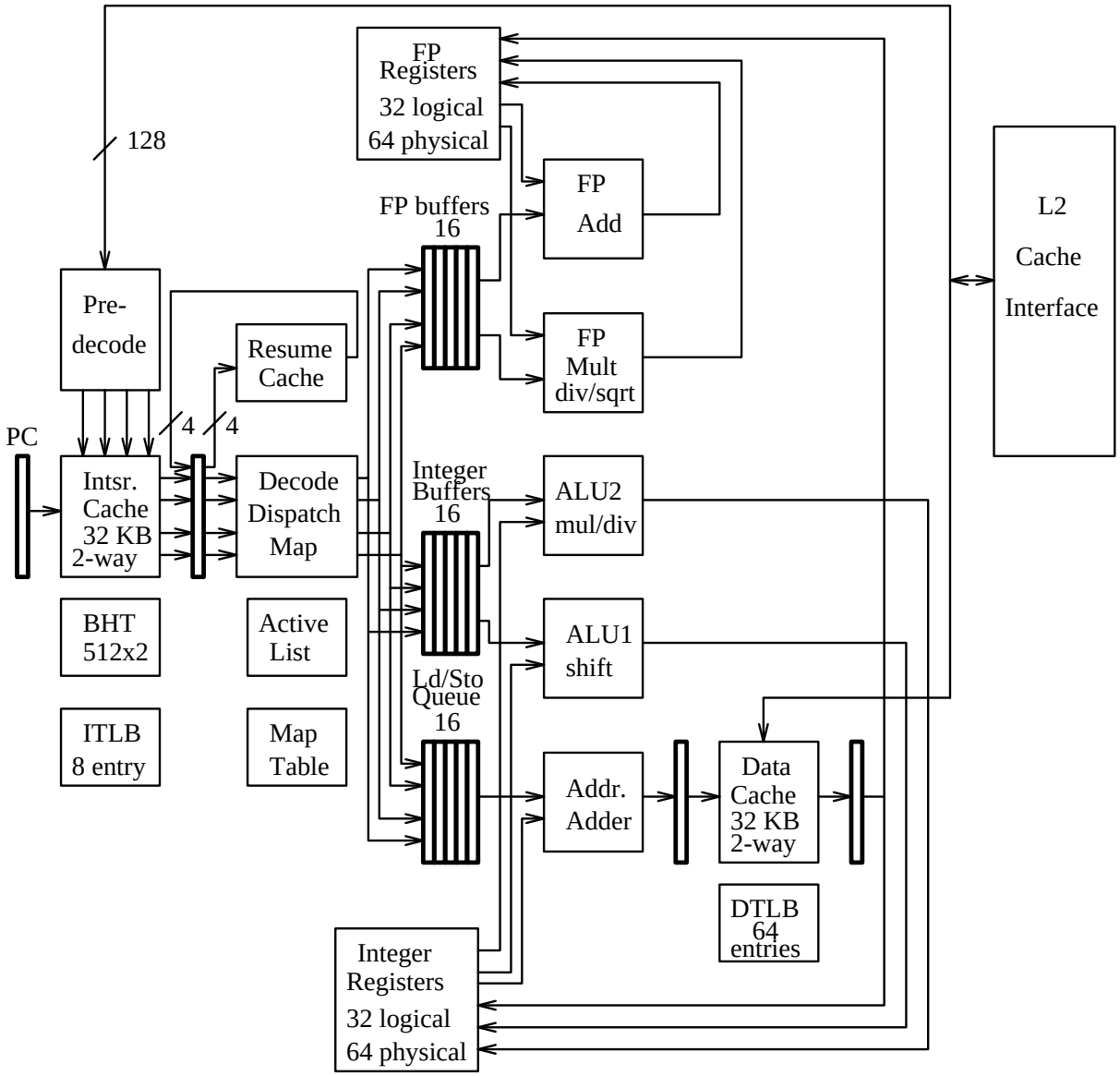
  parallelism expressed within single instructions

  unproven for non-numeric codes

  incomplete--scalar instructions also needed

Superscalar

- A generic superscalar processor
  MIPS R10000



MIPS R10000 contd.

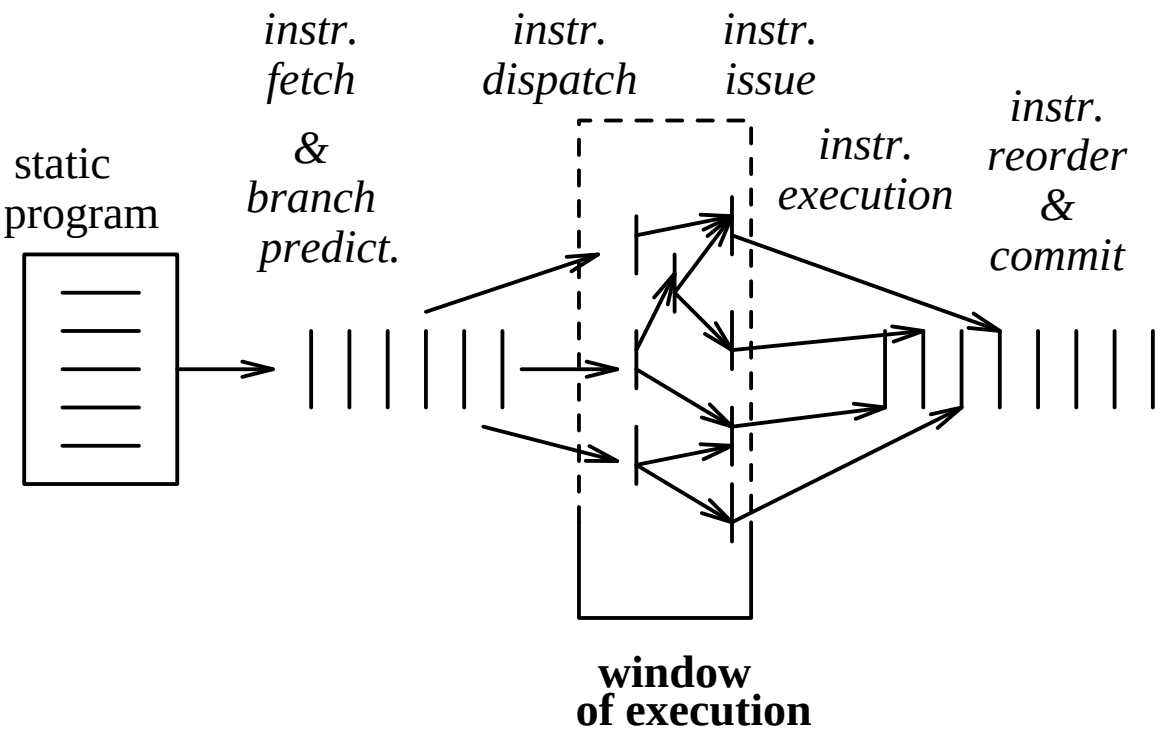
- Single program counter
- Multiple instructions per cycle
- All instructions *dispatched* in order

  to detect branches

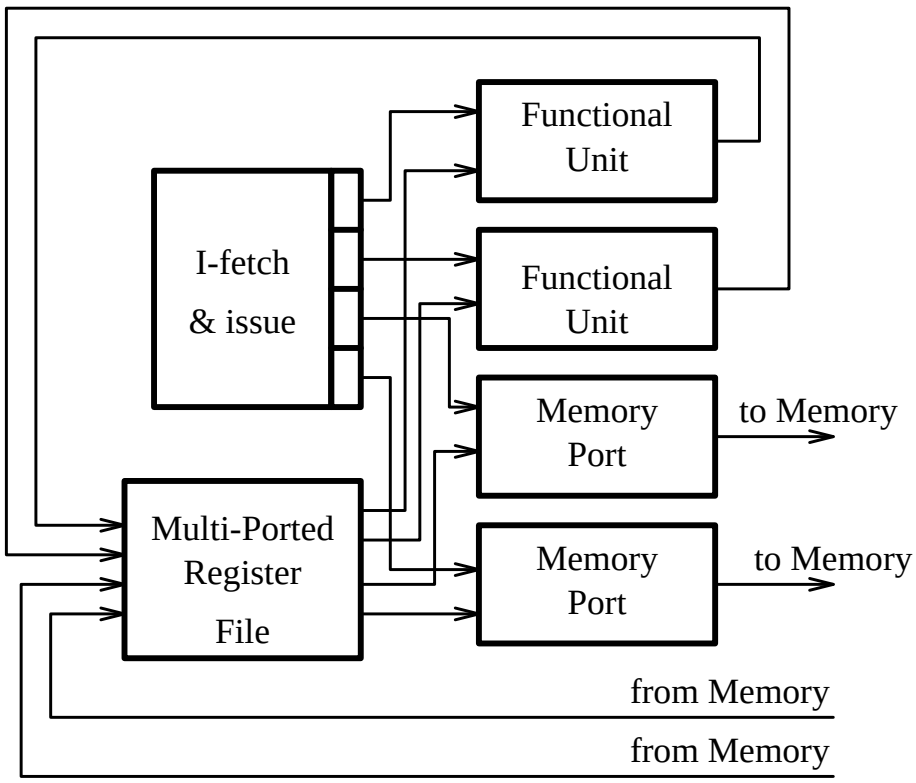
  to detect and resolve dependences
- After dispatch, instructions form a *window*

  held in a collection of buffers and reservation stations
- Instruction issue is dynamic from window
- Memory addresses *calculated* in order but may complete out of order

Superscalar, the Big Picture



VLIW



- Single program counter
- Multiple instructions per cycle from a single, wide instruction word
- Window formed by the compiler while building long instruction words
- Instructions issue in order
- Memory addresses "disambiguated" in software

Limitations of the Current Paradigm(s)

- Issue window size
  - Predictability of conditional branches
  - Issue logic complexity
- Dispatch logic complexity
- Memory hazards
- Centralized resources

Limitation: Issue Window Size

- Consider the dynamic instruction stream

```
| time ->
l l i s b l i b l l f f b l i b l i b l i b l i b l i b l l i i s b l i b l i b l i b
```

l = load instruction
i = integer instruction
f = floating point instruction
b = branch instruction
s = store instruction

- The *window* is defined by oldest and newest instructions that may issue simultaneously
- The wider the window, the more opportunities for parallelism
- In current superscalars, window size is a few 10s
- In VLIW, window can be larger

  Determined by compiler

  But must be something compiler can "see"

  trace scheduling relies on static prediction

  object-oriented program tends to "hide" things

Window size limitations, contd.

- Problem: conditional branches

  Incorrect predictions limit effective window

  In current paradigms, all branches must be predicted as window is formed

- Example:

  .9 prediction accuracy

  => 60% chance of getting past 5 branches
  => at most a few 10s of instructions

Window size limitations, contd.

- Problem: issue logic complexity?

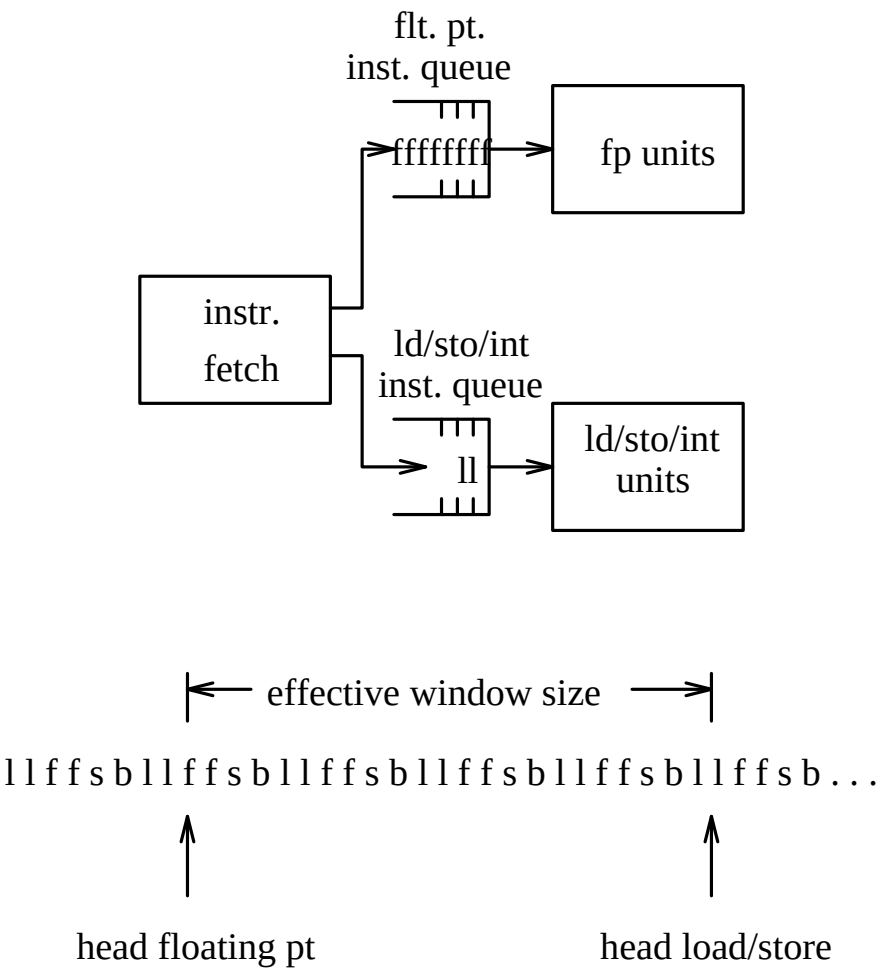
  Alleged n^2 complexity caused by simultaneously inspecting all instructions in window

- Not necessarily:

  A large window does not imply that all instructions are considered for issue every cycle.

- Example: decoupled archs.

Decoupled Arch. Example



- Consider a trace being executed by a decoupled arch.
- The effective window size can be quite large, with only two instructions being considered for issue (the instructions at the queue heads)

Limitations of the Current Paradigm(s)

Limitation: dispatch complexity

- Dependences must be detected and resolved at the maximum throughput rate

  e.g. register re-naming/re-mapping

- This is n^2 complexity and could be an ultimate limitation

  note: ***n*** is peak *dispatch rate*, not window size

Limitations of the Current Paradigm(s)

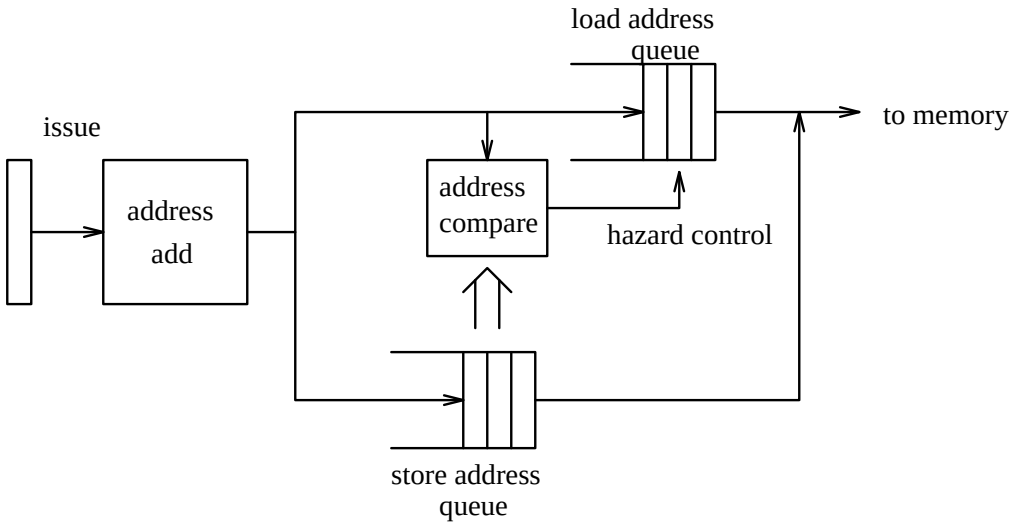
Limitation: memory hazards

- Register hazards are easily detected because there is no indirect addressing or indexing

- Memory hazards cannot always be detected until address is computed.

  => In current superscalars, loads and stores compute addresses in program order

  After addresses are computed,
  loads and stores may pass stores



- In a classic VLIW, loads and stores never pass

Centralized Resources

- Single physical register file
  - => many ports (space complexity)
  - => arbitration (control complexity)
- Single reorder buffer
  - => many ports
- Functional units and cache ports
  - => arbitration

New Paradigms

Key idea: form very large, *accurate* windows by going beyond a single program counter

- Low Overhead Autotasking
- Multiscalar
- Points in Between

Autotasking

- Pioneered by Cray Research
- Compiler-detected parallelism
- Usually loop-oriented
- Shared registers for fast scheduling
- Performance:

  To set up a parallel task: 80+ insts.

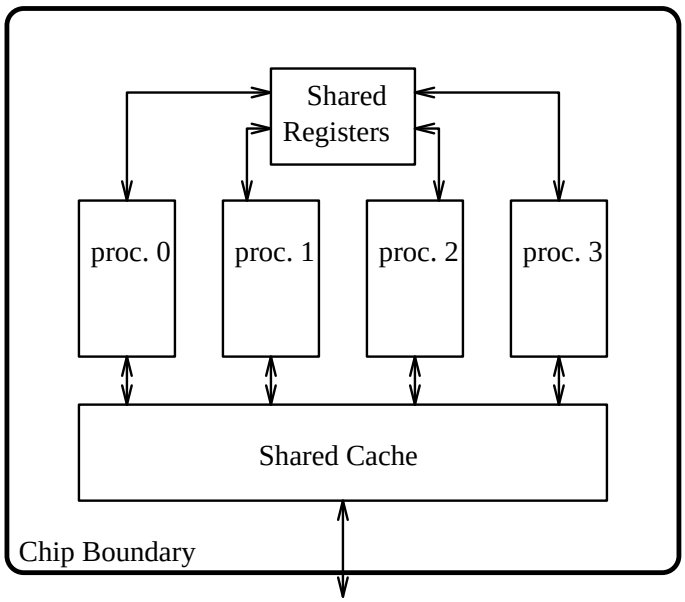
  For processor to leave wait loop: 50+ insts.

  To schedule an iteration: 7 insts.

  => 100s of insts of work needed to amortize overhead

Low Overhead Autotasking

- Goal: cut overheads at least an order of magnitude
- Multiprocessor on a chip



- Traditional goals of MP are scalability, generality, flexibility

  Forget it! these should not be a high priority...
  Pretend the MP is one processing "unit"

- Get scalability generality, flexibility at higher levels.

Low Overhead Autotasking, contd.

- Example: Compiler can assign specific tasks to specific processors to enhance cache locality.

```
Do i= 1,n
  .
   a(i) = ...
  .
End do
  .
  .
Do j = 1,n
  .
       = a(j)
  .
End do
```

Low Overhead Autotasking, contd.

- Big Question: will it work for non-numeric codes?

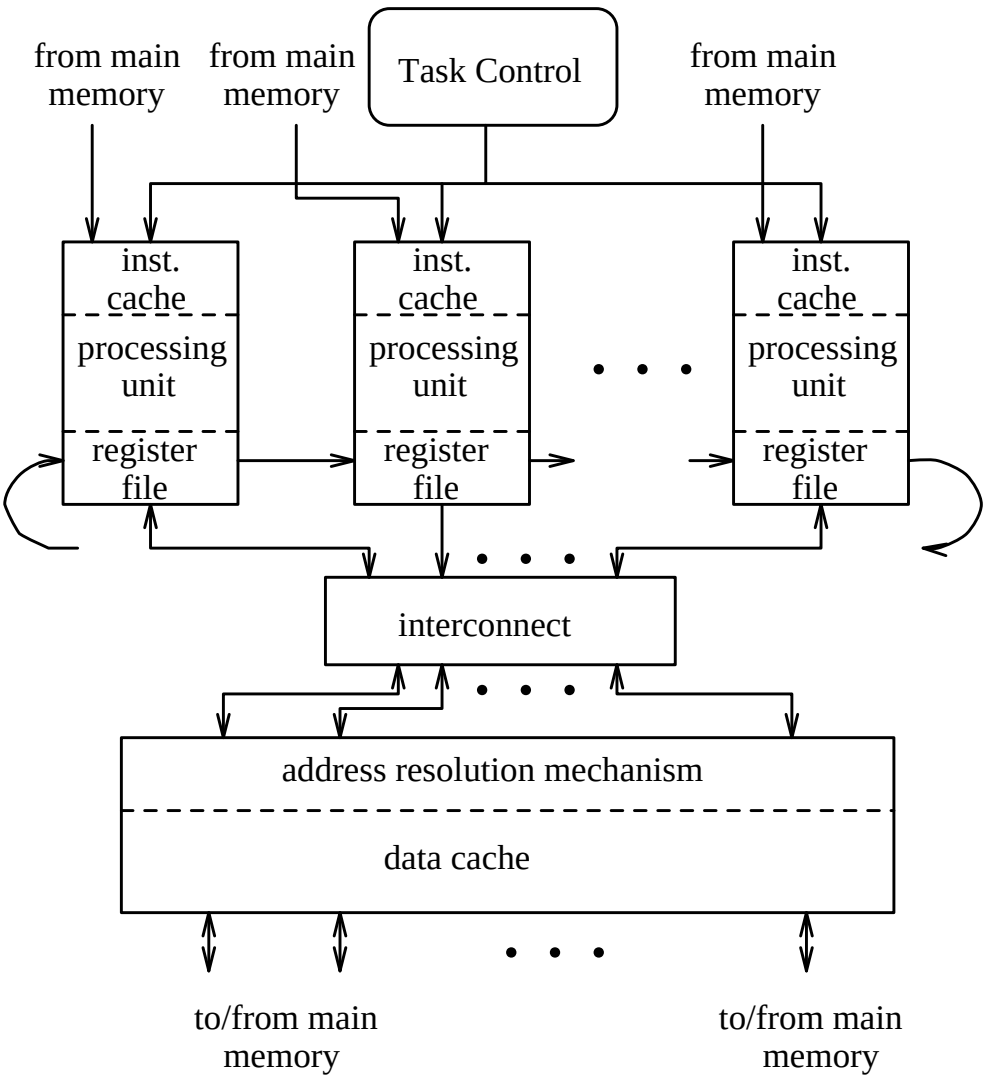
  Will parallelism detection be any more successful when MPs are on a chip?

  Smaller overheads => smaller grain size
  But is that enough??

I shot an arrow in the air...

it fell to earth I know not where

Multiscalar Paradigm



Multiscalar Paradigm, contd.

- Idea: compiler forms tasks

  Task control speculatively allocates tasks in some sequential order

  Hardware (with some compiler help) sorts out the data communication

- Processing units may be superscalar or VLIW

- Tasks complete and commit state in FIFO order

- Processing units are arranged in a ring (a circular queue of processors)

- Loads and stores done speculatively prior to address comparisons

- Incorrect speculation "rolls back" processor queue

Register Communication

- Compiler indicates only the registers that are live when a task is exited (create mask)
- Task control accumulates create masks and passes info to new tasks (accumulate mask)
- All registers in the accumulate mask are reserved when the new task starts
- Values are passed around the register communication ring;

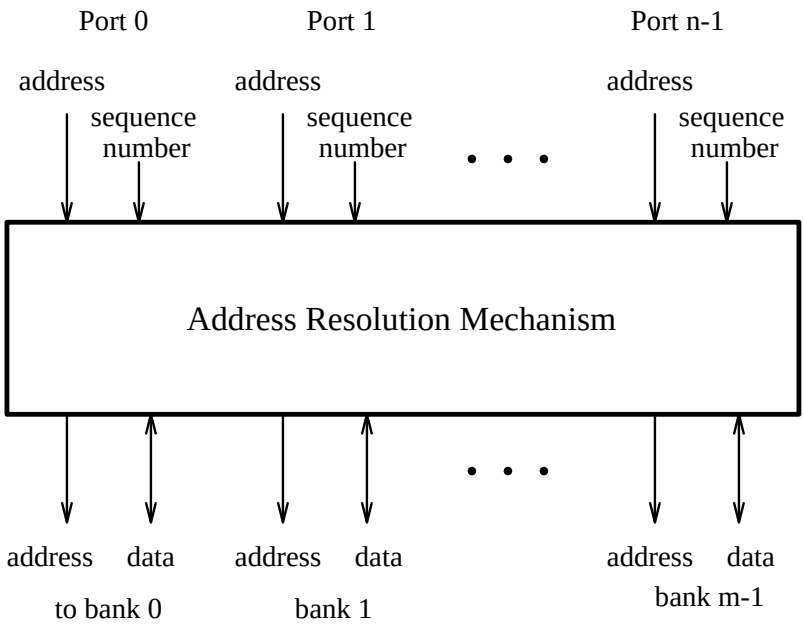
  reservations are removed as registers get updated

Memory Data Communication

- Intra-task hazards resolved in the normal way
- Loads and stores are performed by tasks speculatively and independently
- Inter-task hazards are detected later with rollback if necessary

Inter-task Memory Hazard Detection

- An address resolution mechanism compares and buffers addresses from different processing units.



- Franklin and Sohi "Address Resolution Buffer"
- "Sequenced Cache"

Sequenced Cache

- A cache line contains a time sequence number
- There can be multiple copies of cache data with different sequence numbers
- Sequence numbers allow data forwarding and hazard detection
- Lines are held in cache until task at head of processor queue completes and releases them

Sequenced Cache

- Example:

  Cache Line:

tag	sequence number	read	write	data

  Cache Set Contents:

47	4	1	0	data0
32	6	1	1	data1
47	6	0	1	data2
47	8	1	0	data3

  Event/Action:

  load tag 47 seq. no. 5 action: use data0

  load tag 47 seq. no. 6 action: use data2

  store tag 47 seq no. 7 action: squash 8 and higher

- Key point: organized by address first, then by time

More Realistic Example

- Consider linked list search:

```
/** bufhd is the head of the buffer **/
for (bufptr = bufhd; bufptr ; bufptr = NEXTBUF(bufptr))
    symbol = SYMVAL(bufptr); /* get new symbol **/
    /** Do a search for the symbol in list **/
    for (list = LHD(listhd); list; list = LNEXT(list))
    {   /** if symbol is present, process and quit **/
       if (symbol == LELE(list))
       {
          process(list);
          break;
       }
    }   /** if symbol not found in list, add to head **/
    addlist(LHD(listhd), symbol);
}
```

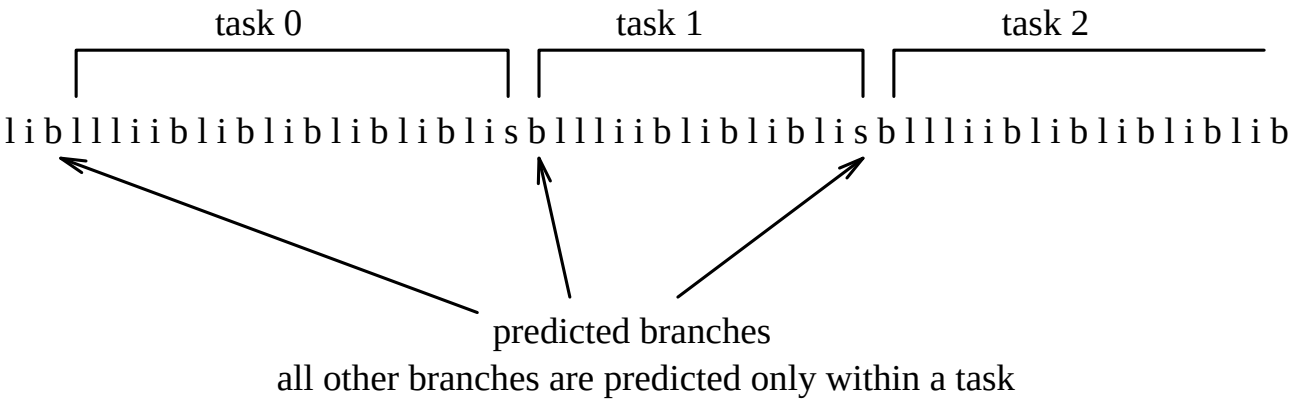
- Each task is a complete list search
- Searches are usually independent and parallel

  Multiscalar can assume they are always independent

- Branches that separate tasks are predictable
- Branches within a task will not be 100% predicted

Summary: Features of New Paradigms

- Multiple prog. counters => large effective windows
- Branch accuracy problem can be reduced:
  embed hard-to-predict branches within a task
- Example:



Summary: Other Issues

- Memory hazards probably biggest problem

  Autotasking: can the compiler detect them?

  Multiscalar: can the hardware handle them (efficiently)?

- n^2 dispatch problem reduced/eliminated by using multiple tasks and register ring

- Physical resources are distributed

  Multiple physical register files each with a small set of ports

  Multiple reorder buffers

  Multiple sets of functional units arbitrated independently

Summary: Points Between

- Low overhead autotasking and multiscalar are two endpoints of a spectrum.
- Another possibility:

  software schedule multiscalar as in autotasking

- And another:

  do register data passing via software, memory data passing in hardware

Conclusions

- Both methods use multiple program counters
  - avoids sequential instruction dispatch
  - avoids predicting all branches
  - => opens up much larger effective windows.
- Autotasking: relatively conservative
  - Technology may open new possibilities by reducing overheads
  - Don't most of the old problems remain?
  - Will it work for non-numeric problems?
- Multiscalar: very aggressive.
  - lots of speculation in both control and data
  - finds parallelism at runtime, by assuming it, then recovering if wrong.
  - good for non-numeric problems
  - load balancing critical in current proposals
  - memory hazard resolution methods unproven (but fundamental)
- Acknowledgement: Guri Sohi and *Kestrel* Multiscalar group

Web site: http://www.cs.wisc.edu/~mscalar/