Multiscalar Processors

Guri Sohi

Computer Sciences Department
University of Wisconsin — Madison
URL: http://www.cs.wisc.edu/~mscalar

Outline

- Wish lists for future processors
- ILP basics
- Multiscalar basics and details
- Preliminary performance results
- Summary and concluding remarks

Hardware Wish List

- Use of simple, regular hardware structures
- Clock speeds comparable to single-issue processors
- Easy growth path from one generation to next
  - Reuse existing processing cores to extent possible
  - No centralized bottlenecks
- Exploit available parallelism

Software Wish List

- Write programs in ordinary languages (e.g. C or C++)
- Target uniform hardware-software interface
  - Facilitate software independence and growth path
- Maintain uniform hardware-software interface, i.e., do not tailor for specific architecture
  - Minimal OS impact
  - Facilitate hardware independence and growth path
- Place few demands on software
  - make minimum requirements for guarantees

The Opportunity and Objective

- Many tens of millions of transistors on a chip vs. few million today

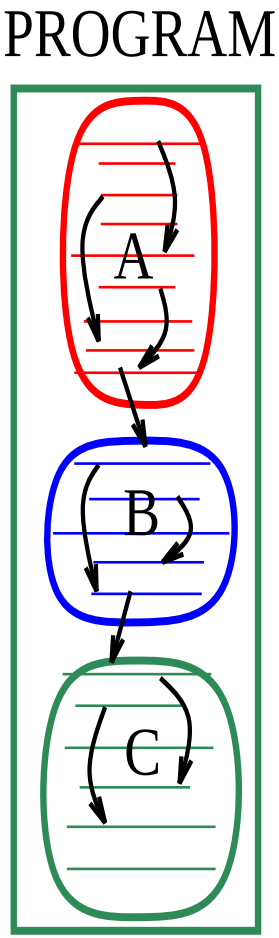
- Can integrate several (tens?) of todays processors, plus supporting hardware, on a chip

Use available resources to minimize program execution time!

A Bird's Eye View

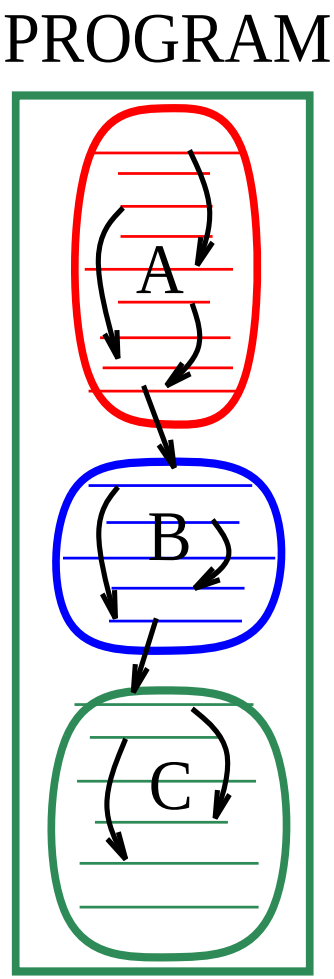
- Start with a static representation of a program
- Sequence through the program to generate the dynamic stream of operations
  - Use single PC to walk through static representation
- Execute operations in dynamic stream as quickly as is possible

Speed up this entire process



ILP Basics

- Sequence through static representation and establish a window of execution
- Establish dependence relationships within window
- Set up parallel execution schedule for operations in window
- Provide sufficient resources to implement parallel execution schedule



Target: 10 IPC

- Establish and maintain a large window (100s of instructions)
- Initiate at least 10 operations into this window per cycle
- Provide lots of storage for inter-operation communications
- Provide means for flexible operation movement in window

Superscalar/VLIW Paradigm

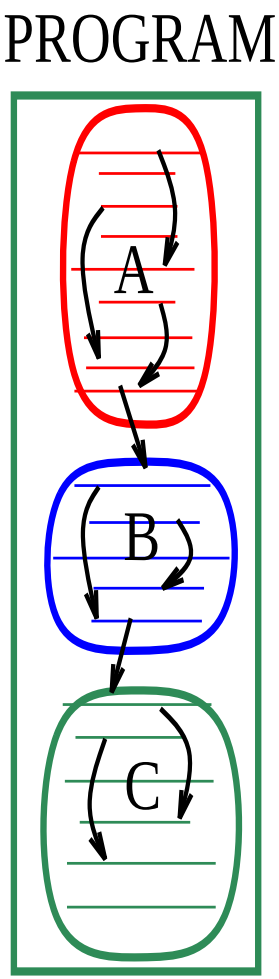
- Use a "single" PC to sequence through static program, "instruction by instruction"
- Establish a contiguous window of operations
  Branch prediction accuracy limits size
- Set up dependence relationships
  Complex decoder hardware in superscalar
- Schedule execution of N independent operations per cycle
  Centralized resources to implement schedule

Multiscalar Paradigm

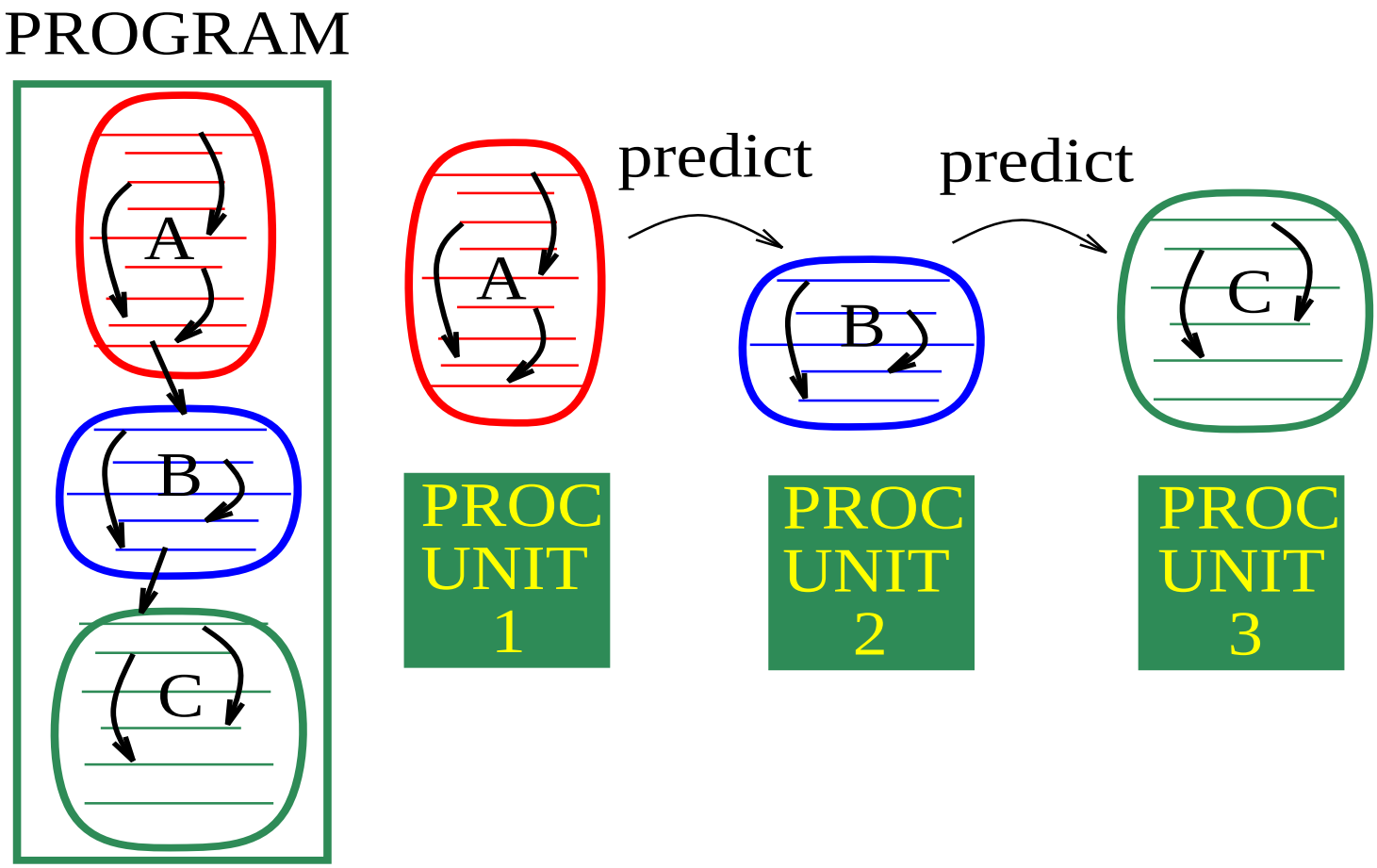
- Break sequencing process into two steps
  - Sequence through static representation in *task-sized* steps
  - Sequence through each task in conventional manner
- Split large instruction window into ordered tasks
- Assign a task to a simple execution engine; exploit ILP by overlapping execution of multiple tasks
- Use separate PCs to sequence through separate tasks
- Maintain the appearance of a single-PC sequencing through the static representation

What is a Task?

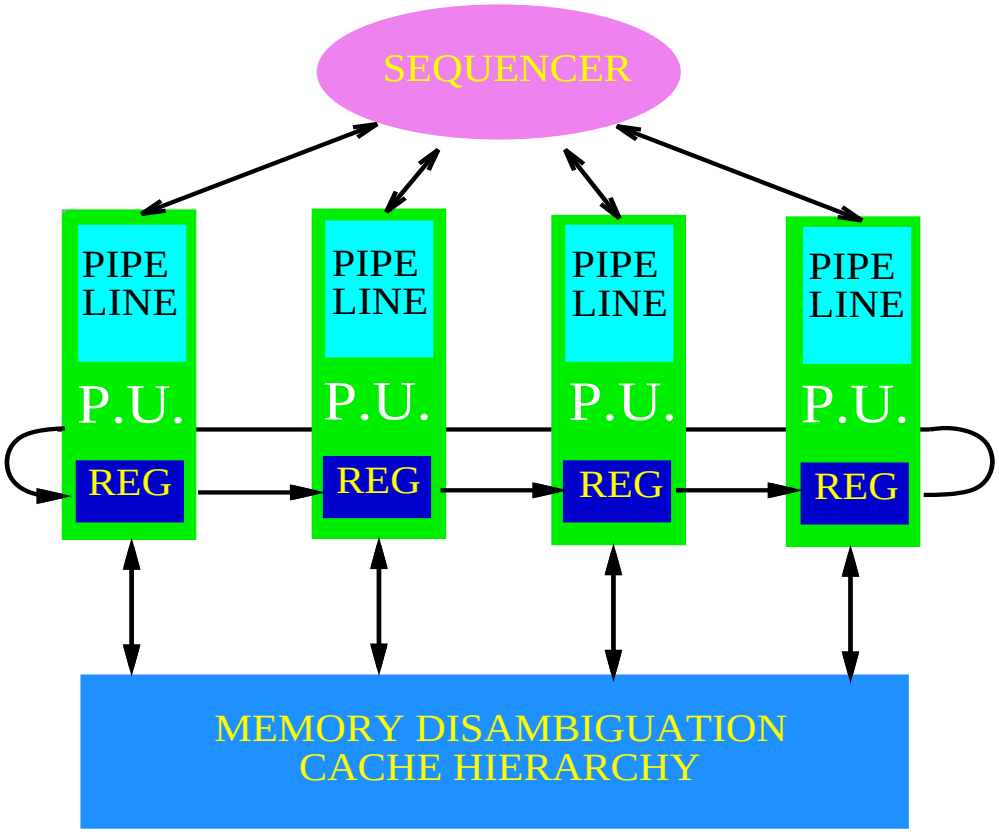
- A portion of the static representation resulting in a contiguous portion of the dynamic instruction stream
  - part of a basic block
  - basic block
  - multiple basic blocks
  - loop iteration
  - entire loop
  - procedure call, etc



Multiscalar Big Picture: Basics



Multiscalar Big Picture: Hardware



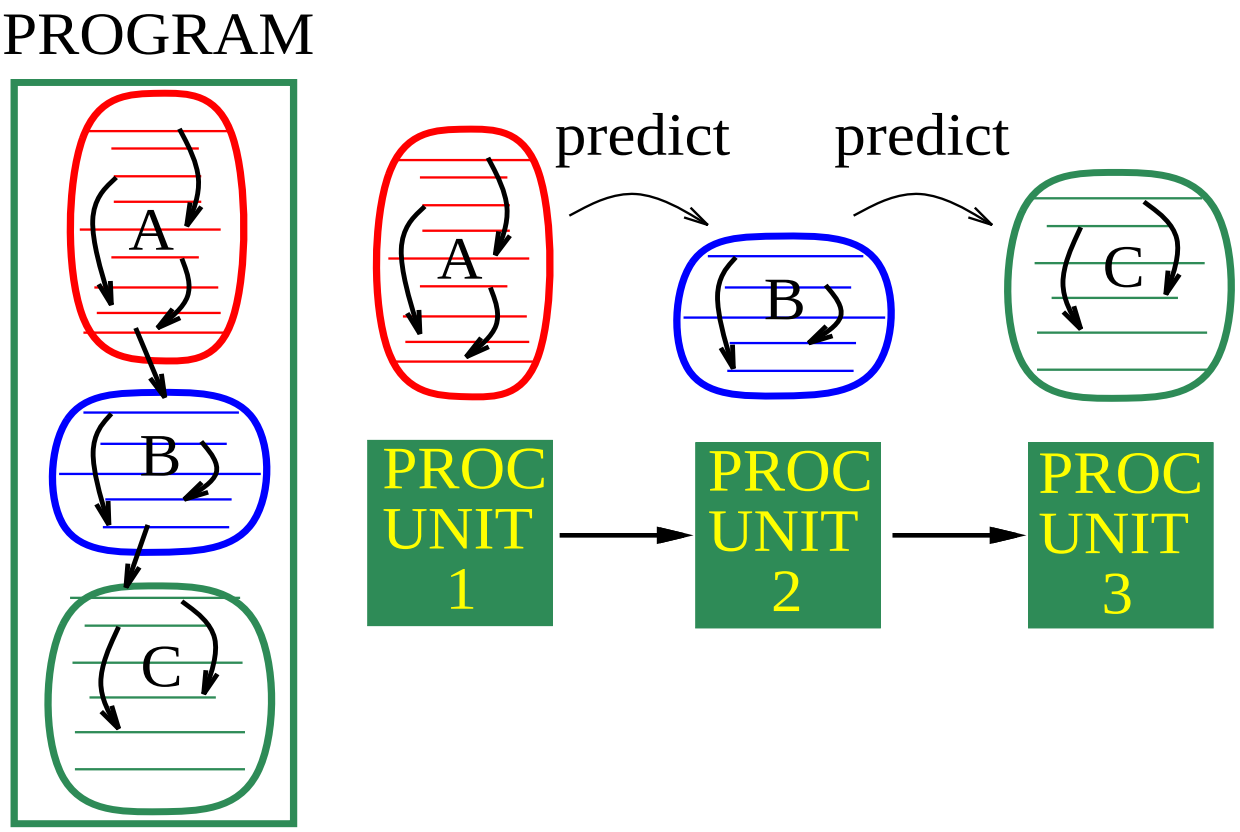
More on Multiscalar Hardware

- Processor consists of several processing cores (or units)
  - each core executes a task
  - each core is equivalent to a typical datapath
- Execution cores are connected in a logical order (queue)
  - hardware pointers to head and tail
  - share logical register and memory address spaces
- ***Active*** cores (ones between head and tail)
  - contain tasks in logical (sequential) order
  - together constitute a large dynamic window

More on Multiscalar Hardware

- Tasks complete and commit (logical) state in FIFO order
- Incorrect speculation “rolls back” queue
  - Control speculation
  - Data speculation

Multiscalar Big Picture: Control

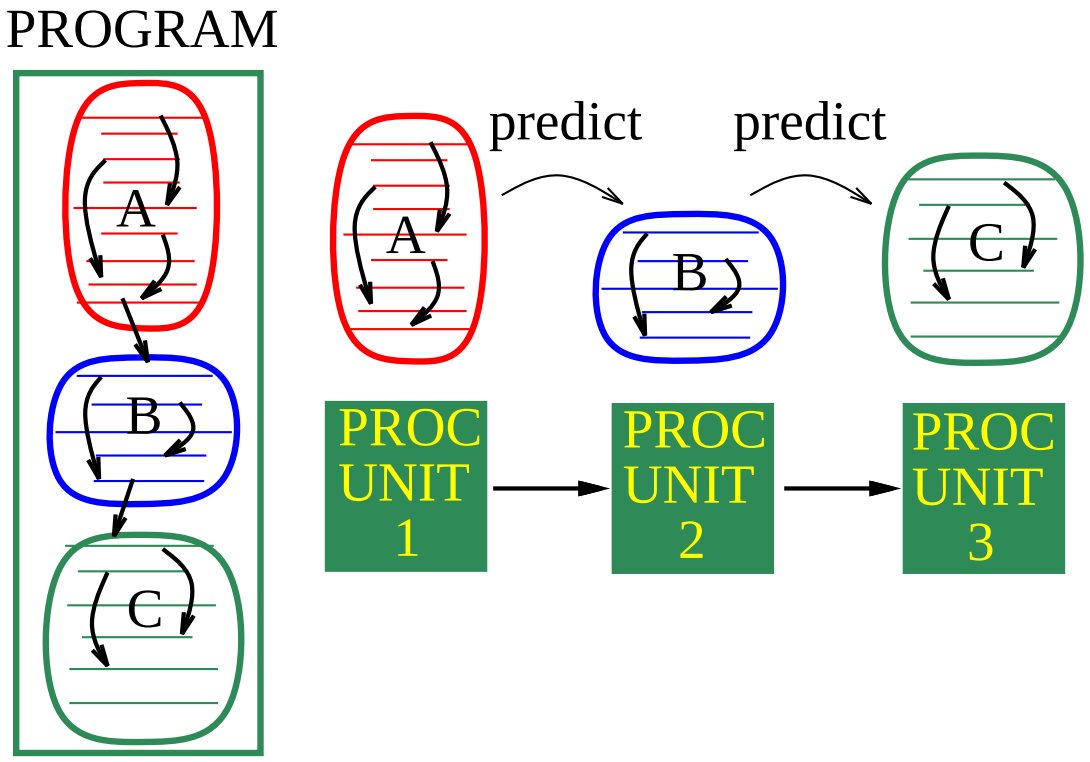


More on Control

- Sequencer manages processing cores like a queue
- Sequencer opens up “instruction window” by (speculatively) assigning a new task to the tail unit
  - Tells core to execute task starting at a given PC
  - Does not perform instruction fetching and decoding
- Intra-task control controls execution of task
- Intra-task control determines when control flows out of task (i.e., task is complete)
- Intra-task branches do not affect creation of “instruction window”

Big Picture: Data Values

- Tasks produce and consume data values bound to registers and memory locations



Issues in Managing Data Values

- Storage
  - values can be produced speculatively
  - where should values be buffered?
- Synchronization
  - ensure that an instruction in a task uses value created by the logical predecessor
- Communication
  - forwarding a value created by an instruction to all future instructions that might need it
- Sequential ordering of tasks influences answers to above

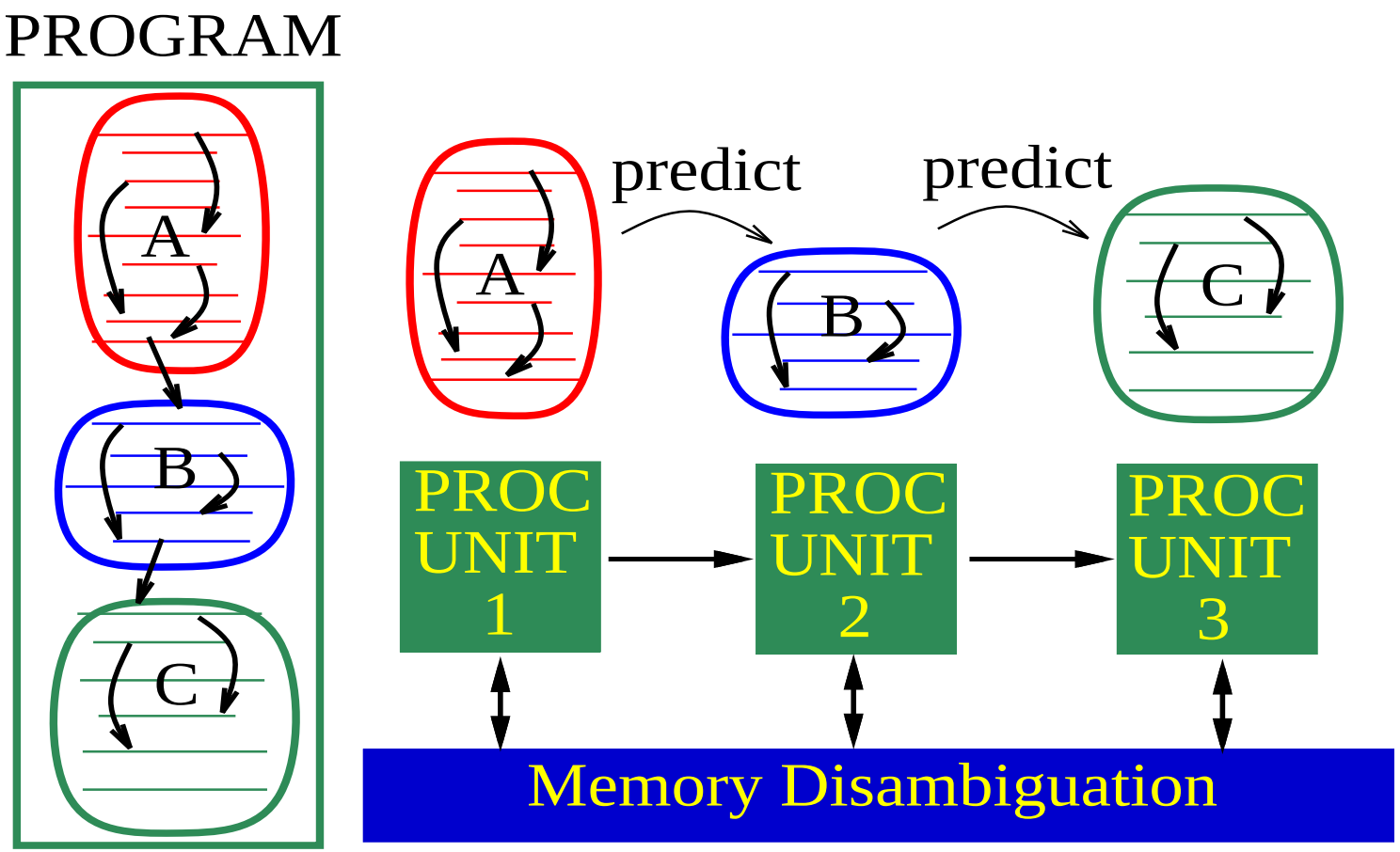
Register Values

- Each core works out of its “local” register file
- Multiple register files act like separate “renamed” files
- Each register file contains register state at a particular time in the (speculative) execution of a program

Register Values

- Can be analyzed statically
- Compiler indicates registers that may be live when task is exited *(create mask)*
- Task control accumulates create masks and passes info to new tasks *(accumulate mask)*
- Registers in the accumulate mask are *reserved* when a new task starts
- Values are *forwarded* on the register communication ring
  - reservations are removed as registers get updated

Memory Values



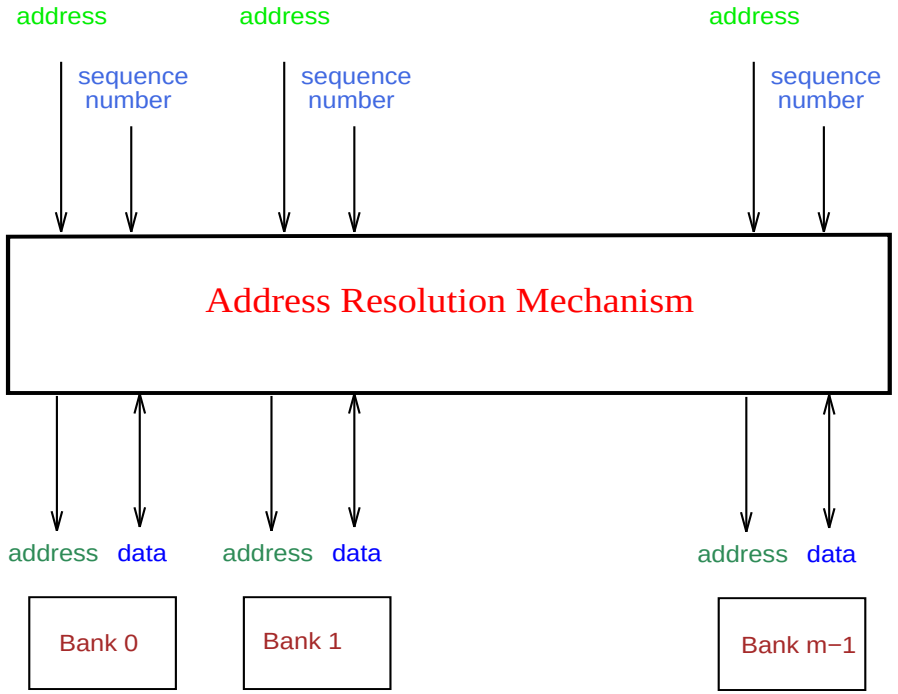
Memory Values

- Can't be analyzed statically
  - can't use bits to synchronize all memory operations
- Can't allow loads in succeeding tasks to wait for (all) stores in preceeding tasks to be resolved
- Perform loads (stores) speculatively, i.e., *data speculation*
  - provide storage for speculative values
- Violation of dependences occurs if store in preceeding task occurs later in time than a load in a successor task
- Provide means to detect violations of dependences, and roll back if necessary

More on Memory Values

- An address resolution mechanism compares and buffers addresses from different processing cores
- Each processing core has a (speculative) image of memory at a different time during the execution of the program
  - Allows memory renaming
- Arbitrary order of memory operations possible
- Arbitrary speculation of memory operations possible

Address Resolution Mechanism



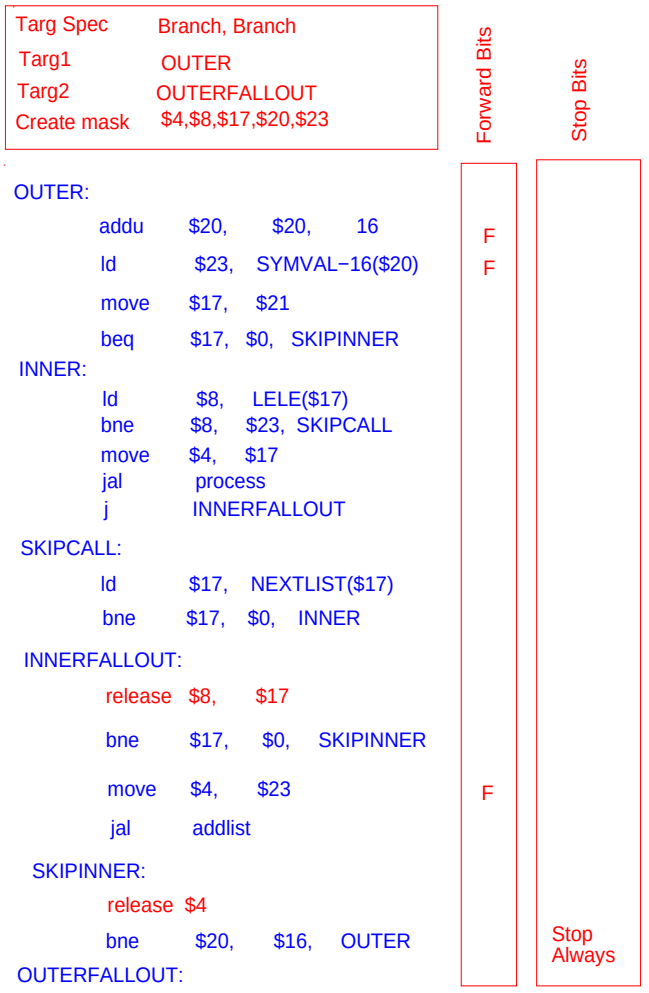
- Address Resolution Buffer (ARB)
- Temporal or sequenced cache/buffers

Multiscalar Programs

- Divide static program into tasks
- For each task, determine:
  - Possible successor tasks
  - How control flows out of task
  - Values created by task
    - values bound to registers
    - values bound to memory

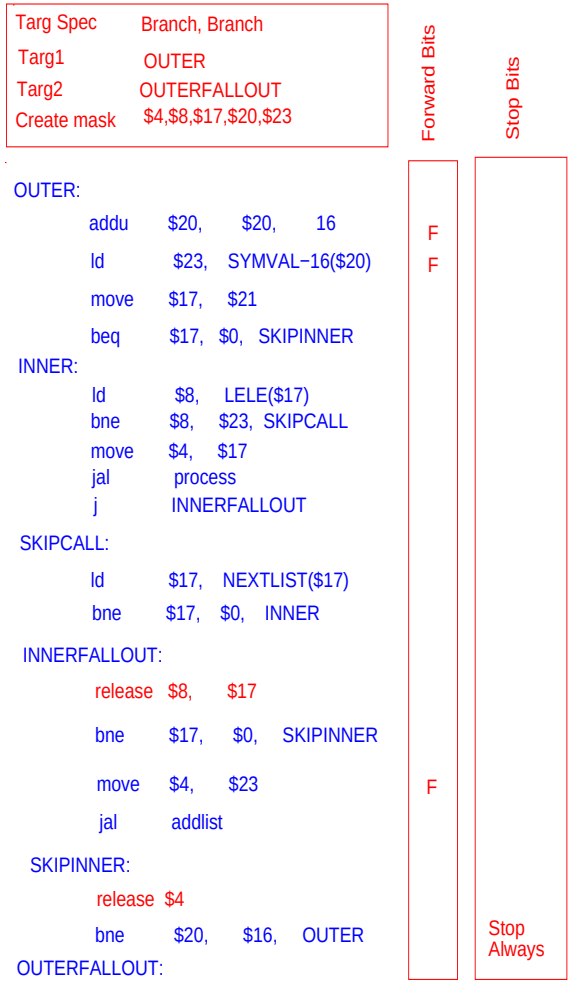
Problems and their Solutions

- Which task to execute after current task?
  - *task prediction* by sequencer
  - similar to branch prediction
- When should a later task wait for a value?
  - register *create mask*
- When should a task forward a value?
  - register *forward bits*
- When is a task over?
  - task *stop bits*



Problems and their Solutions

- Conservative create mask (due to control flow in task)
  - *Release* instructions
- Our-of-order memory operations
  - Address resolution mechanism



Example: Problem

- Process stream of tokens
- Create entry in list for new token
- Use information in list to process token

Example: C Code

```
for (indx = 0; indx < BUFSIZE; indx++) {
    /* get the symbol for which to search */
    symbol = SYMVAL(buffer[indx]);

    /* do a linear search fo rthe symbol in the list */
    for (list = listhd; list; list = LNEXT(list) {
        /* if symbol already present, process entry */
        if (symbol == LELE(list)) {
            process(list);
            break;
        }
     }

     /* if symbol not found, add it to the tail */
    if (! list) {
        addlist(symbol);
    }
}
```

Example

- 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% predictable
  - Superscalar/VLIW will not be able to overlap processing of different tokens

Example: Executable

Targ Spec	Branch, Branch
Targ1	OUTER
Targ2	OUTERFALLOUT
Create mask	$4,$8,$17,$20,$23

```
OUTER:
        addu    $20,    $20,    16
        ld      $23,    SYMVAL−16($20)
        move    $17,    $21
        beq     $17,    $0,     SKIPINNER
INNER:
        ld      $8,     LELE($17)
        bne     $8,     $23,    SKIPCALL
        move    $4,     $17
        jal     process
        j       INNERFALLOUT
SKIPCALL:
        ld      $17,    NEXTLIST($17)
        bne     $17,    $0,     INNER
INNERFALLOUT:
        release $8,     $17
        bne     $17,    $0,     SKIPINNER
        move    $4,     $23
        jal     addlist
SKIPINNER:
        release $4
        bne     $20,    $16,    OUTER
OUTERFALLOUT:
```

Forward Bits	Stop Bits
F	
F	
F	
	Stop Always

Going from one generation to another could leave binary untouched!

Binary Compatibility Options

- Multiscalar-specific information (task successors, create masks, forward bits, stop bits) is available in a binary
- Recover information at run time
  - "Low" performance but run ordinary binaries
- Binary to binary translation
  - Better performance by including some optimizations
- Compiler
  - Best performance, but needs recompilation

Regardless, binary from one multiscalar generation to another can remain the same

Comparison with Multiprocessors

Attributes	Multiprocessor	Multiscalar
Speculative task initiation	No/Difficult	Yes
Multiple flows of control	Yes	Yes
Task determination	Static	Static (possibly dynamic)
Software guarantee of inter-task control independence	Required	Not required
Software knowledge of inter-task data dependences	Required	Not required
Inter-task sync.	Explicit	Implicit/Explicit
Inter-task communication	Through memory Through messages	Through registers and memory
Register space	Distinct for PEs	Common for PEs
Memory space	Common Distinct	Common for PEs

Making Multiscalar Tick

- Tasks are predicted well, or incorrect predictions known soon
  - Can get around hard-to-predict branches by including in task
- Tasks are “large enough” to overcome pipeline oveheads
- Tasks are of equal dynamic length, else load balancing problem
- Task is scheduled for efficient execution on processing core
- Inter-task dependences are scheduled properly
- Memory dependences are not violated often

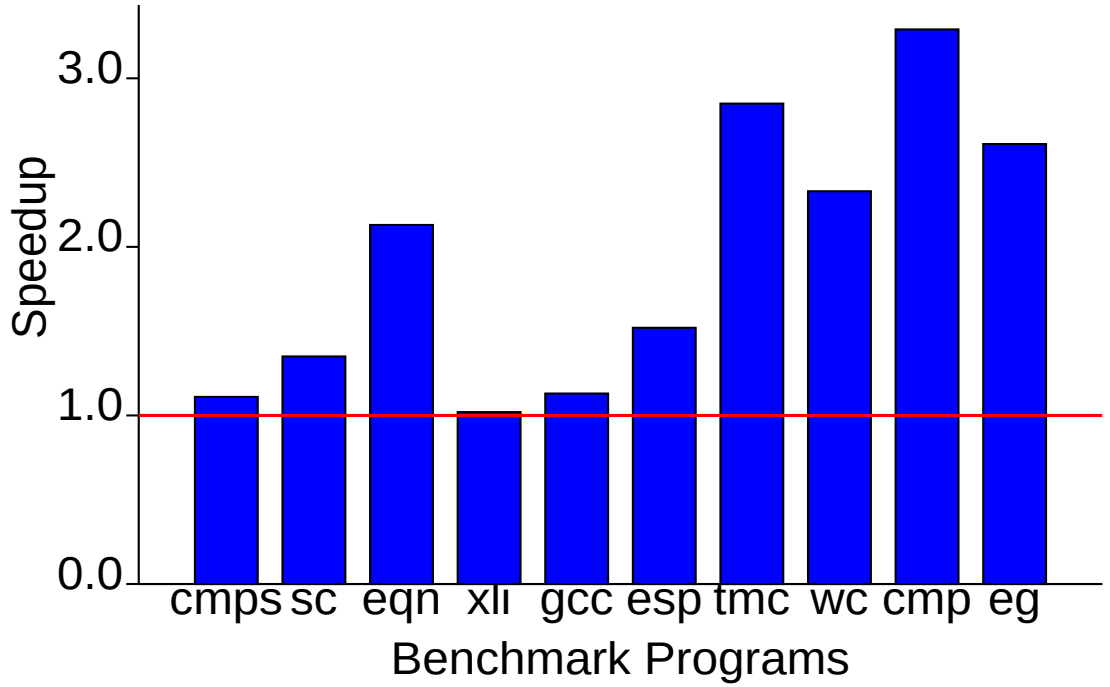
Current Status

- Project in existence for 4-5 years
  - 1-2 students per year
- GCC-based compiler
  - can generate multiscalar executable for arbitrary C program
  - naive task selection and scheduling
  - no memory disambiguation used
- Detailed timing simulator
  - accepts executable and carries out a cycle-by-cycle simulation of its execution, varying core capabilities
- Initiated large scale effort (Kestrel)

Current Performance Results

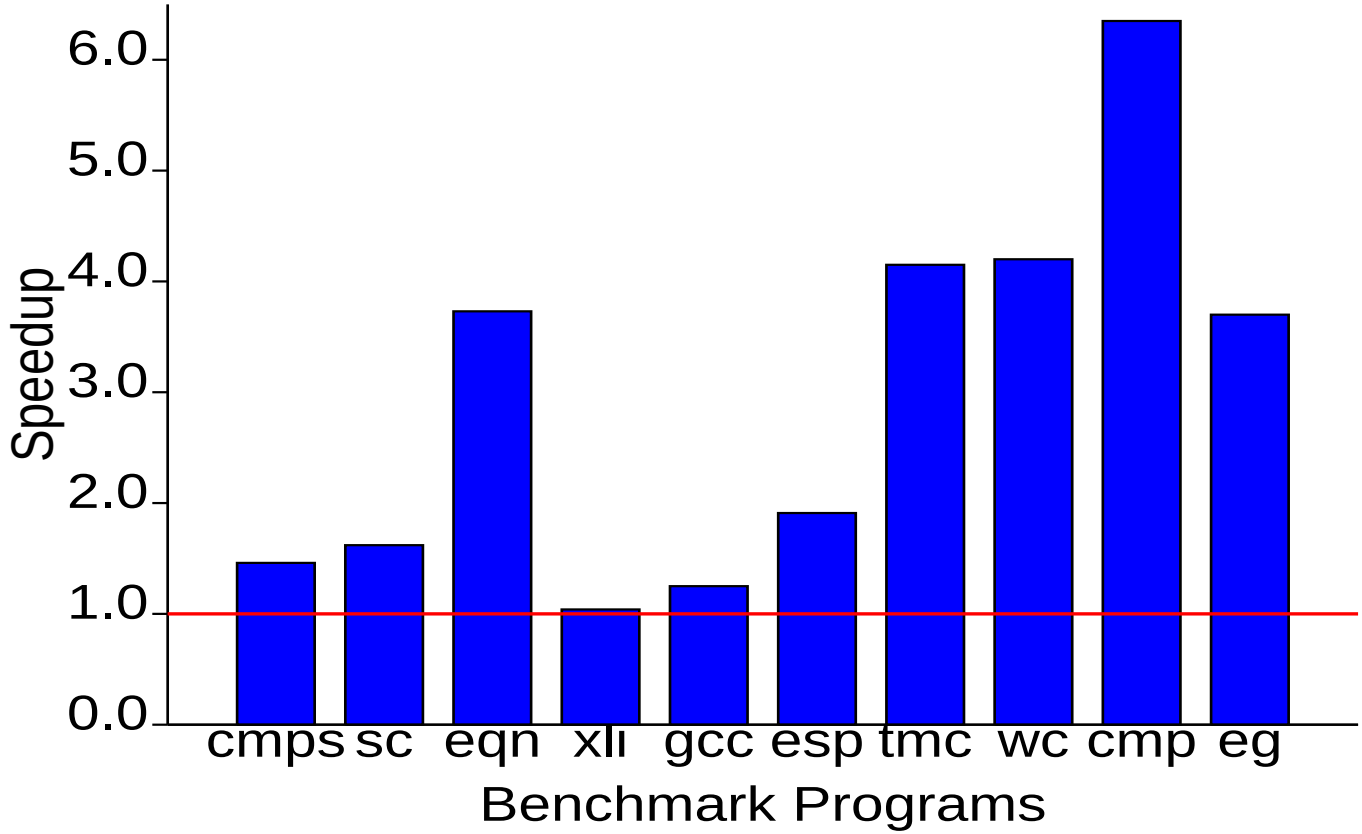
Four processing cores

Approx. 15% extra instructions, plus extra cycle for cache hit



Current Performance Results

Eight processing cores



Multiscalar Kestrel Project

- Demonstrate feasibility of both hardware and software
  - Hardware
    - 8 moderately superscalar processing cores
    - based upon MIPS-IV ISA
  - Software
    - integrated (front- and back-end) compiler
- Funded by ARPA and NSF
- Collaborative effort between Wisconsin and Minnesota
  - Wisconsin PIs: Sohi and Smith
  - Minnesota collaborators: Yew, Li, and Lilja

Ongoing/Planned Work

- Hardware design and simulation at multiple levels
  - Clock level (for performance evaluation)
  - Verilog functional level for non-multiscalar specific (e.g., floating point)
  - Verilog/Synopsys gate-level for multiscalar specific
  - Circuit level for special functions (e.g., ARB)
- Very accurate performance and hardware cost estimates
- Integrated compilers
  - integrate front end (SUIF) and back end (GCC)
  - better task selection algorithms and heuristics
  - inter- and intra-task scheduling algorithms

Ongoing/Planned Work

- Utility for C++, database, and other non-numeric programs
- Alternate microarchitectures
- Alternate memory disambiguation mechanisms and hardware

Concluding Remarks

- Superscalar has some life left
  - different ways of looking at it may give it even more life
- If compiler has full knowledge about all dependence relationships, use a Multiprocessor, with very fine-grain synchronization
- If full knowledge is not available, use Multiscalar
- Multiscalar platform allows both!