Multiscalar Processors

Gurindar S. Sohi, Scott E. Breach, T.N. Vijaykumar

Computer Sciences Department University of Wisconsin-Madison

ISCA 1995

Presented by Benjamin Gundersen

Outline

Executive Summary

Instruction-level Parallelism

Goal

Multiscalar Paradigm

Multiscalar Hardware

Analyzing CPU cycles

Comparison to other ILPs

Performance Evaluation

Strengths and Weaknesses

Inspired Work

Discussion

Outline

Executive Summary

Instruction-level Parallelism

Goal

Multiscalar Paradigm

Multiscalar Hardware

Analyzing CPU cycles

Comparison to other ILPs

Performance Evaluation

Strengths and Weaknesses

Inspired Work

Discussion

Executive Summary

- **Problem:** Improving performance of sequential execution is critical for modern systems.

Executive Summary

- **Problem:** Improving performance of sequential execution is critical for modern systems.
- **Goal:** Execute many instructions in parallel per cycle.

Executive Summary

- **Problem:** Improving performance of sequential execution is critical for modern systems.
- **Goal:** Execute many instructions in parallel per cycle.
- **Key idea:** Introduce the Multiscalar Paradigm where each Program is divided into a collection of tasks to increase instruction level parallelism.

Executive Summary

- **Problem:** Improving performance of sequential execution is critical for modern systems.
- **Goal:** Execute many instructions in parallel per cycle.
- **Key idea:** Introduce the Multiscalar Paradigm where each Program is divided into a collection of tasks to increase instruction level parallelism.
- **Mechanism:** Each task is distributed to one of many parallel processing units while using one logical register file.

Executive Summary

- **Problem:** Improving performance of sequential execution is critical for modern systems.
- **Goal:** Execute many instructions in parallel per cycle.
- **Key idea:** Introduce the Multiscalar Paradigm where each Program is divided into a collection of tasks to increase instruction level parallelism.
- **Mechanism:** Each task is distributed to one of many parallel processing units while using one logical register file.
- **Result:** Multiscalar processor greatly improve performance in parallelisable workloads.

Outline

Executive Summary

Instruction-level Parallelism

Goal

Multiscalar Paradigm

Multiscalar Hardware

Analyzing CPU cycles

Comparison to other ILPs

Performance Evaluation

Strengths and Weaknesses

Inspired Work

Discussion

Instruction-level Parallelism (ILP)

- **Pipelining:** Execution of multiple instructions can partially overlap.

Instruction-level Parallelism (ILP)

- **Pipelining:** Execution of multiple instructions can partially overlap.
- **Superscalar:** Fetch and dispatch multiple instructions at once.

Instruction-level Parallelism (ILP)

- **Pipelining:** Execution of multiple instructions can partially overlap.
- **Superscalar:** Fetch and dispatch multiple instructions at once.
- **Very Long Instruction Word:** (VLIW) Encode multiple instructions in one instruction.

Instruction-level Parallelism (ILP)

- **Pipelining:** Execution of multiple instructions can partially overlap.
- **Superscalar:** Fetch and dispatch multiple instructions at once.
- **Very Long Instruction Word:** (VLIW) Encode multiple instructions in one instruction.
- **Out-of-order:** Instructions execute in any order that does not violate data dependencies.

Instruction-level Parallelism (ILP)

- **Pipelining:** Execution of multiple instructions can partially overlap.
- **Superscalar:** Fetch and dispatch multiple instructions at once.
- **Very Long Instruction Word:** (VLIW) Encode multiple instructions in one instruction.
- **Out-of-order:** Instructions execute in any order that does not violate data dependencies.
- **Dataflow:** Instructions execute once input is available.

Key Constraint of previous mechanisms

- Many instructions are independent of each other. Sequential execution does not exploit independent instructions.

Key Constraint of previous mechanisms

- Many instructions are independent of each other. Sequential execution does not exploit independent instructions.
- Previous mechanisms focused on increasing ILP, like VLIW or Superscalar exhibit a **key constraint:** stall instructions until all previous control dependencies have been resolved.

Outline

Executive Summary

Instruction-level Parallelism

Goal

Multiscalar Paradigm

Multiscalar Hardware

Analyzing CPU cycles

Comparison to other ILPs

Performance Evaluation

Strengths and Weaknesses

Inspired Work

Discussion

Goal

- Increase ILP without the constraint of stalling until all previous control dependencies have been resolved by proposing the **Multiscalar Paradigm**.

Outline

Executive Summary
Instruction-level Parallelism
Goal
Multiscalar Paradigm
Multiscalar Hardware
Analyzing CPU cycles
Comparison to other ILPs
Performance Evaluation
Strengths and Weaknesses
Inspired Work
Discussion

Multiscalar Paradigm; Key Idea

- Cooperation between Software and Hardware.

Multiscalar Paradigm; Key Idea

- Cooperation between Software and Hardware.
- Split the program into **tasks** using the control flow graph.

Multiscalar Paradigm; Key Idea

- Cooperation between Software and Hardware.
- Split the program into **tasks** using the control flow graph.
- **Speculatively distribute** tasks in to parallel processing units to extract ILP.

Multiscalar Paradigm; Key Idea

- Cooperation between Software and Hardware.
- Split the program into **tasks** using the control flow graph.
- **Speculatively distribute** tasks in to parallel processing units to extract ILP.
- **Pass values** between processing units.

Multiscalar Paradigm; Key Idea

- Cooperation between Software and Hardware.
- Split the program into **tasks** using the control flow graph.
- **Speculatively distribute** tasks in to parallel processing units to extract ILP.
- **Pass values** between processing units.
- Impose **sequential appearance** by constraining when instructions can be executed.

Multiscalar Paradigm; Outline

- Possible Hardware Implementation

Multiscalar Paradigm; Outline

- Possible Hardware Implementation
- Control Flow Graph

Multiscalar Paradigm; Outline

- Possible Hardware Implementation
- Control Flow Graph
- Definition of Task

Multiscalar Paradigm; Outline

- Possible Hardware Implementation
- Control Flow Graph
- Definition of Task
- Imposing Sequential Appearance

Multiscalar Paradigm; Outline

- Possible Hardware Implementation
- Control Flow Graph
- Definition of Task
- Imposing Sequential Appearance
- Example Code

Multiscalar Paradigm; Outline

- Possible Hardware Implementation
- Control Flow Graph
- Definition of Task
- Imposing Sequential Appearance
- Example Code
- Multiscalar Program

Multiscalar Paradigm; Outline

- Possible Hardware Implementation
- Control Flow Graph
- Definition of Task
- Imposing Sequential Appearance
- Example Code
- Multiscalar Program
- Example Program

Possible Hardware Implementation

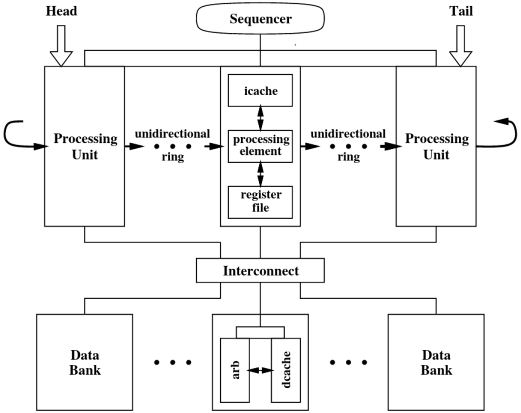


Figure: Example Hardware

Control Flow Graph (CFG)

- CFG consists of basic blocks (nodes) and control flow (edges).

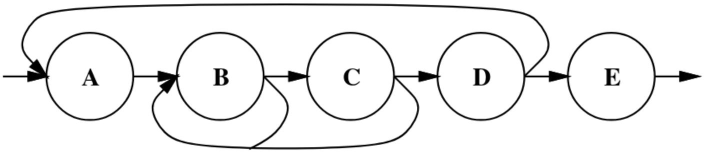


Figure: Control Flow Graph.

Control Flow Graph (CFG)

- CFG consists of basic blocks (nodes) and control flow (edges).
- First instruction of basic block is the entry point (unique).

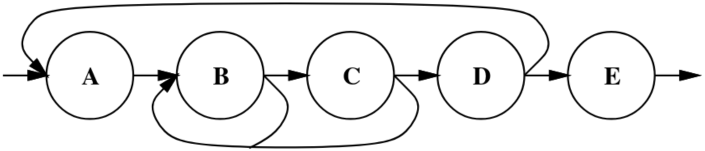


Figure: Control Flow Graph.

Control Flow Graph (CFG)

- CFG consists of basic blocks (nodes) and control flow (edges).
- First instruction of basic block is the entry point (unique).
- Last instruction is the **only control flow instruction** in a basic block.

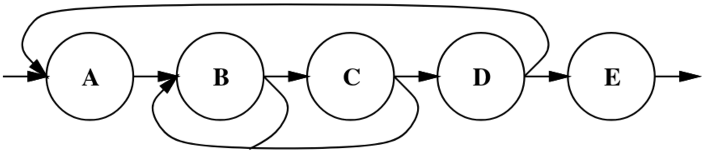


Figure: Control Flow Graph.

Definition of Task

- Task is a portion of CFG.

Definition of Task

- Task is a portion of CFG.
- Corresponds to a **contiguous region** of a dynamic instruction sequence. (Examples: part of basic block, single loop iteration, function call, multiple basic blocks).

Definition of Task

- Task is a portion of CFG.
- Corresponds to a **contiguous region** of a dynamic instruction sequence. (Examples: part of basic block, single loop iteration, function call, multiple basic blocks).
- Tasks are assigned to processing units for execution.

Definition of Task

- Task is a portion of CFG.
- Corresponds to a **contiguous region** of a dynamic instruction sequence. (Examples: part of basic block, single loop iteration, function call, multiple basic blocks).
- Tasks are assigned to processing units for execution.
- Tasks are not independent of each other.

Imposing Sequential Appearance

- **Challenge of Multiscalar Paradigm:** Ensure that each processing unit adheres to **sequential execution semantics**.

Imposing Sequential Appearance

- **Challenge of Multiscalar Paradigm:** Ensure that each processing unit adheres to **sequential execution semantics**.
- Now we will look at these critical factors to impose sequential order:

Imposing Sequential Appearance

- **Challenge of Multiscalar Paradigm:** Ensure that each processing unit adheres to **sequential execution semantics**.
- Now we will look at these critical factors to impose sequential order:
- **Processing Unit order**

Imposing Sequential Appearance

- **Challenge of Multiscalar Paradigm:** Ensure that each processing unit adheres to **sequential execution semantics**.
- Now we will look at these critical factors to impose sequential order:
- **Processing Unit order**
- **Passing Values:** Register and Memory Synchronization

Imposing Sequential Appearance

- **Challenge of Multiscalar Paradigm:** Ensure that each processing unit adheres to **sequential execution semantics**.
- Now we will look at these critical factors to impose sequential order:
- **Processing Unit order**
- **Passing Values:** Register and Memory Synchronization
- **Speculative Tasks**

Imposing Sequential Appearance

- **Challenge of Multiscalar Paradigm:** Ensure that each processing unit adheres to **sequential execution semantics**.
- Now we will look at these critical factors to impose sequential order:
- **Processing Unit order**
- **Passing Values:** Register and Memory Synchronization
- **Speculative Tasks**
- **Task Retirement**

Order on Processing Units

- Enforce loose sequential order over all processing units. Which imposes sequential order on tasks.

Order on Processing Units

- Enforce loose sequential order over all processing units. Which imposes sequential order on tasks.
- Organize units in **circular queue**.

Order on Processing Units

- Enforce loose sequential order over all processing units. Which imposes sequential order on tasks.
- Organize units in **circular queue**.
- Head an tail pointers indicate which units are executing earliest and last of the current tasks.

Passing values

- Executing instructions in a task **produce and consume values**.

Passing values

- Executing instructions in a task **produce and consume values**.
- Values are either bound to a location in memory or to registers.

Passing values

- Executing instructions in a task **produce and consume values**.
- Values are either bound to a location in memory or to registers.
- In multiscalar execution there are multiple PUs, the view of one **single set of registers and memory locations** must be upheld.

Passing values

- Executing instructions in a task **produce and consume values**.
- Values are either bound to a location in memory or to registers.
- In multiscalar execution there are multiple PUs, the view of one **single set of registers and memory locations** must be upheld.
- Produced and consumed values must be the **same as in sequential execution**.

Passing values

- Executing instructions in a task **produce and consume values**.
- Values are either bound to a location in memory or to registers.
- In multiscalar execution there are multiple PUs, the view of one **single set of registers and memory locations** must be upheld.
- Produced and consumed values must be the **same as in sequential execution**.
- **Solution:** Register and Memory synchronization.

Register synchronization

- In the Multiscalar Paradigm register values which a task may produce can be statically determined.

Register synchronization

- In the Multiscalar Paradigm register values which a task may produce can be statically determined.
- Produced values in a task are forwarded to successor tasks.

Register synchronization

- In the Multiscalar Paradigm register values which a task may produce can be statically determined.
- Produced values in a task are forwarded to successor tasks.
- Consuming instructions have to **wait for all values it wants to consume**.

Memory synchronization

- **Memory locations known:** similar approach to registers.

Memory synchronization

- **Memory locations known:** similar approach to registers.
- **Not known:** Either take conservative approach or aggressive approach.

Memory synchronization

- **Memory locations known:** similar approach to registers.
- **Not known:** Either take conservative approach or aggressive approach.
- **Conservative:** Wait until it is certain that a load will read correct value.

Memory synchronization

- **Memory locations known:** similar approach to registers.
- **Not known:** Either take conservative approach or aggressive approach.
- **Conservative:** Wait until it is certain that a load will read correct value.
- **Aggressive:** Loads are performed speculatively. Conflicts must be resolved.

Memory synchronization

- **Memory locations known:** similar approach to registers.
- **Not known:** Either take conservative approach or aggressive approach.
- **Conservative:** Wait until it is certain that a load will read correct value.
- **Aggressive:** Loads are performed speculatively. Conflicts must be resolved.
- Multiscalar processors take the **aggressive approach**.

Speculative Tasks

- Task may be **speculative** because of control speculation (branch prediction) or data speculation.

Speculative Tasks

- Task may be **speculative** because of control speculation (branch prediction) or data speculation.
- If a conflict occurs the task and all successors must be squashed.

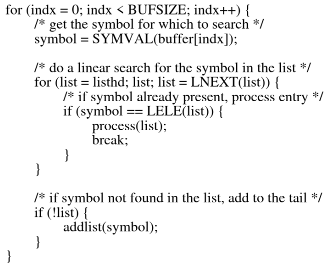
Task Retirement

- **Only when retirement of a task is imminent the values produced by the task are certain**.

Task Retirement

- **Only when retirement of a task is imminent the values produced by the task are certain**.
- Since values are forwarded earlier tasks must be retired in the order they were added.

Example Code

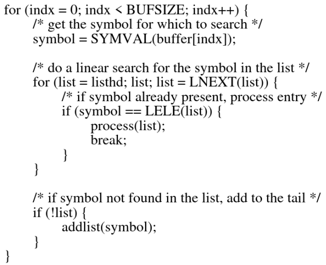


```
for (indx = 0; indx < BUFSIZE; indx++) {
        /* get the symbol for which to search */
        symbol = SYMVAL(buffer[indx]);

        /* do a linear search for the 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 in the list, add to the tail */
        if (!list) {
                addlist(symbol);
        }
}
```

Figure: Example Code Segment

- Take symbol from buffer and if it is in list process it. Otherwise add it to the list.

Example Code

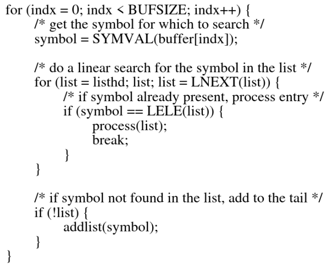


```
for (indx = 0; indx < BUFSIZE; indx++) {
    /* get the symbol for which to search */
    symbol = SYMVAL(buffer[indx]);

    /* do a linear search for the 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 in the list, add to the tail */
    if (!list) {
        addlist(symbol);
    }
}
```

Figure: Example Code Segment

- Take symbol from buffer and if it is in list process it. Otherwise add it to the list.
- Assumption: After running for a while most symbols will already be in the list. Thus list is not updated frequently.

Example Code



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

        /* do a linear search for the 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 in the list, add to the tail */
        if (!list) {
                addlist(symbol);
        }
}
```

Figure: Example Code Segment

- Take symbol from buffer and if it is in list process it. Otherwise add it to the list.
- Assumption: After running for a while most symbols will already be in the list. Thus list is not updated frequently.
- List not changing much means that many tasks can run independently from each other. Thus we get an execution of multiple instructions per cycle.

Multiscalar Paradigm; Next Steps

- Multiscalar Programs

Multiscalar Paradigm; Next Steps

- Multiscalar Programs
- Sequencer

Multiscalar Paradigm; Next Steps

- Multiscalar Programs
- Sequencer
- Communication between tasks

Multiscalar Paradigm; Next Steps

- Multiscalar Programs
- Sequencer
- Communication between tasks
- Example Program

Multiscalar Programs

- Must enable **fast walk through CFG** to distribute tasks on many processing units.

Multiscalar Programs

- Must enable **fast walk through CFG** to distribute tasks on many processing units.
- Contains actual code, CFG structure and communication characteristics.

Multiscalar Programs

- Must enable **fast walk through CFG** to distribute tasks on many processing units.
- Contains actual code, CFG structure and communication characteristics.
- Only minimal changes have to be made to the ISA, thus an existing ISA can be used as basis.

Sequencer

- Assigns tasks to processing units.

Sequencer

- Assigns tasks to processing units.
- Needs to know successors of tasks.

Sequencer

- Assigns tasks to processing units.
- Needs to know successors of tasks.
- Chooses **one possible successor task** to continue the CFG walk.

Sequencer

- Assigns tasks to processing units.
- Needs to know successors of tasks.
- Chooses **one possible successor task** to continue the CFG walk.
- Controlflow information can be statically determined and is placed in a task descriptor.

Sequencer

- Assigns tasks to processing units.
- Needs to know successors of tasks.
- Chooses **one possible successor task** to continue the CFG walk.
- Controlflow information can be statically determined and is placed in a task descriptor.
- Task descriptor may be placed within program text or in a single location

Communication between tasks

- Only last update of a register in task should forward to successor tasks.

Communication between tasks

- Only last update of a register in task should forward to successor tasks.
- Not all execution paths update all values. Non updated values must also be communicated.

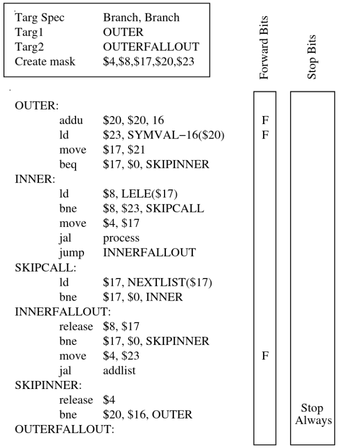
Communication between tasks

- Only last update of a register in task should forward to successor tasks.
- Not all execution paths update all values. Non updated values must also be communicated.
- Instructions which possibly leave the task are known.

Communication between tasks

- Only last update of a register in task should forward to successor tasks.
- Not all execution paths update all values. Non updated values must also be communicated.
- Instructions which possibly leave the task are known.
- **The compiler is our friend** and can solve these problems for us.

Example Program

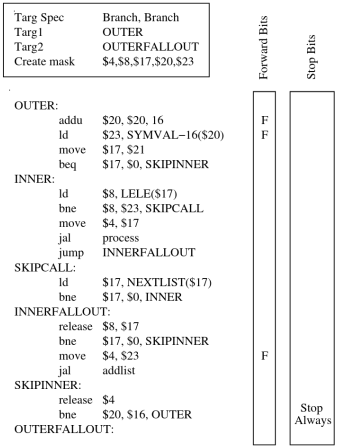


```
Targ Spec       Branch, Branch
Targ1           OUTER
Targ2           OUTERFALLOUT
Create mask     $4,$8,$17,$20,$23

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
        jump    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:
```

Figure: Example Program

- Task creates values bound to registers: 4, 8, 17, 20, 23

Example Program

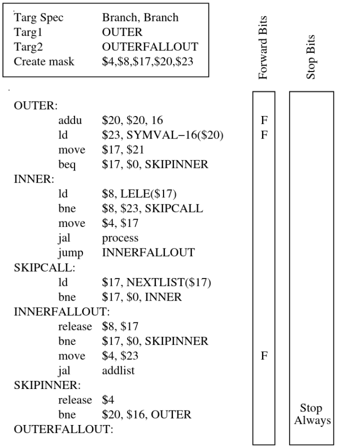


```
Targ Spec      Branch, Branch
Targ1          OUTER
Targ2          OUTERFALLOUT
Create mask    $4,$8,$17,$20,$23

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
        jump     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:
```

Figure: Example Program

- Task creates values bound to registers: 4, 8, 17, 20, 23
- 18, 17 must be released after loop since they are repeatedly updated in the loop.

Example Program

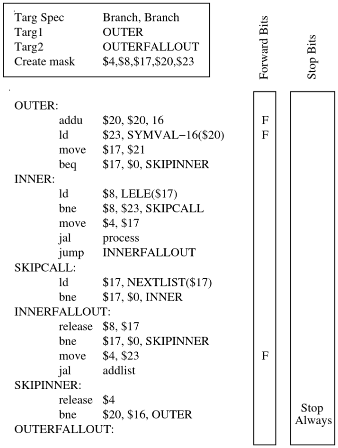


```
Targ Spec      Branch, Branch
Targ1          OUTER
Targ2          OUTERFALLOUT
Create mask    $4,$8,$17,$20,$23

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
        jump     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:
```

Figure: Example Program

- Task creates values bound to registers: 4, 8, 17, 20, 23
- 18, 17 must be released after loop since they are repeatedly updated in the loop.
- The other values have forward bits.

Example Program



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

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

Figure: Example Program

- Task creates values bound to registers: 4, 8, 17, 20, 23
- 18, 17 must be released after loop since they are repeatedly updated in the loop.
- The other values have forward bits.
- 4 is released if its update code is skipped.

Augmenting binaries

- We analyze existing binaries, generate the multiscalar information (CFG, task structure) and add the information to the binary.

Augmenting binaries

- We analyze existing binaries, generate the multiscalar information (CFG, task structure) and add the information to the binary.
- Possible for non multiscalar and multiscalar binaries.

Augmenting binaries

- We analyze existing binaries, generate the multiscalar information (CFG, task structure) and add the information to the binary.
- Possible for non multiscalar and multiscalar binaries.
- Allows to change multiscalar interface by augmenting a binary.

Outline

Executive Summary

Instruction-level Parallelism

Goal

Multiscalar Paradigm

Multiscalar Hardware

Analyzing CPU cycles

Comparison to other ILPs

Performance Evaluation

Strengths and Weaknesses

Inspired Work

Discussion

Multiscalar Hardware; One of many implementations

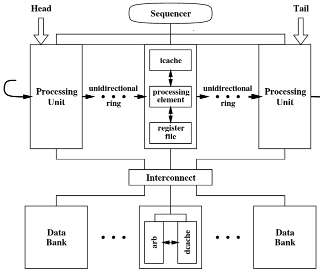


Figure: Example Hardware

Key Components:

- **Sequencer**

Multiscalar Hardware; One of many implementations

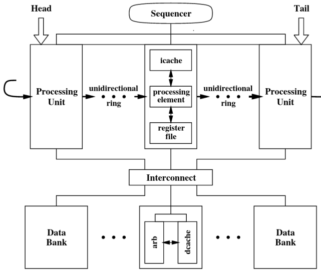


Figure: Example Hardware

Key Components:

- **Sequencer**
- **Processing Units**

Multiscalar Hardware; One of many implementations

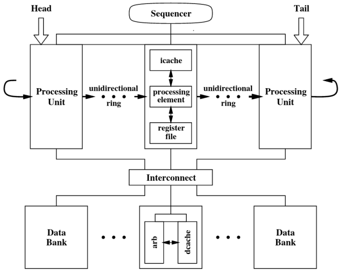


Figure: Example Hardware

Key Components:

- **Sequencer**
- **Processing Units**
- **Data Banks**

Sequencer

- Sequencer decides on order of tasks.

Sequencer

- Sequencer decides on order of tasks.
- Fetches task descriptor and then invokes task.

Sequencer

- Sequencer decides on order of tasks.
- Fetches task descriptor and then invokes task.
- Invocation consists of providing the address of the first instruction of the task, information to enable the passing of values.

Sequencer

- Sequencer decides on order of tasks.
- Fetches task descriptor and then invokes task.
- Invocation consists of providing the address of the first instruction of the task, information to enable the passing of values.
- Given task descriptor determine / predict next task.

Processing Unit

- Processing units **independently fetch and execute instructions** of their assigned task.

Processing Unit

- Processing units **independently fetch and execute instructions** of their assigned task.
- When it encounters a stop bit the condition is evaluated and if it is true then task is completed.

Processing Unit

- Processing units **independently fetch and execute instructions** of their assigned task.
- When it encounters a stop bit the condition is evaluated and if it is true then task is completed.
- Through the unidirectional ring which connects all processing units information is forwarded.

Data Bank

- Data banks consist of cache banks and Address Resolution Buffers (ARB).

Data Bank

- Data banks consist of cache banks and Address Resolution Buffers (ARB).
- ARBs **hold speculative memory operations**, detect memory dependency violations and initiate corrective action.

Data Bank

- Data banks consist of cache banks and Address Resolution Buffers (ARB).
- ARBs **hold speculative memory operations**, detect memory dependency violations and initiate corrective action.
- Cache only updated after speculative values become non speculative values.

Data Bank

- Data banks consist of cache banks and Address Resolution Buffers (ARB).
- ARBs **hold speculative memory operations**, detect memory dependency violations and initiate corrective action.
- Cache only updated after speculative values become non speculative values.
- ARBs track units which performed operations.

Outline

Executive Summary

Instruction-level Parallelism

Goal

Multiscalar Paradigm

Multiscalar Hardware

Analyzing CPU cycles

Comparison to other ILPs

Performance Evaluation

Strengths and Weaknesses

Inspired Work

Discussion

Analyzing CPU cycles

- **Objective:** Each processing unit should perform useful computation. And thus in combination the PUs execute multiple instructions per cycle. What we want to avoid:

Analyzing CPU cycles

- **Objective:** Each processing unit should perform useful computation. And thus in combination the PUs execute multiple instructions per cycle. What we want to avoid:
- **Non-useful computation** because it will be squashed later.

Analyzing CPU cycles

- **Objective:** Each processing unit should perform useful computation. And thus in combination the PUs execute multiple instructions per cycle. What we want to avoid:
- **Non-useful computation** because it will be squashed later.
- Performs **no computation** because task is waiting for values.

Analyzing CPU cycles

- **Objective:** Each processing unit should perform useful computation. And thus in combination the PUs execute multiple instructions per cycle. What we want to avoid:
- **Non-useful computation** because it will be squashed later.
- Performs **no computation** because task is waiting for values.
- Remains **idle** since head is not finished but predecessor task has finished executing all instructions.

How to avoid

- **Non-useful computation:** Synchronization of scalars and globals

How to avoid

- **Non-useful computation:** Synchronization of scalars and globals
- **No computation:** Early Validation of Prediction

How to avoid

- **Non-useful computation:** Synchronization of scalars and globals
- **No computation:** Early Validation of Prediction
- **Idle:** Reduce inter-task dependencies and balance the load.

Non-useful Computation: Synchronization

- **Experience:** Squashes because of memory conflict are usually caused by updates of **global scalars and structures**.

Non-useful Computation: Synchronization

- **Experience:** Squashes because of memory conflict are usually caused by updates of **global scalars and structures**.
- Thus these accesses should be **synchronized**.

No Computation: Early Validation of Prediction

- Catching false prediction **lowers time spent on non-computation cycles significantly**.

No Computation: Early Validation of Prediction

- Catching false prediction **lowers time spent on non-computation cycles significantly**.
- Could change structure of loops such that loop exit test is performed at the beginning.

No Computation: Early Validation of Prediction

- Catching false prediction **lowers time spent on non-computation cycles significantly**.
- Could change structure of loops such that loop exit test is performed at the beginning.
- Could add explicit prediction validation instructions.

Idle: Reduce Inter-Task Dependencies

- **Dependencies may result in near sequential execution**.

Idle: Reduce Inter-Task Dependencies

- **Dependencies may result in near sequential execution**.
- Consider: Induction variable updated as last instruction in a loop versus induction variable updated at beginning of a loop and copy kept for current task.

Idle: Load Balancing

- Some tasks may have a lot less work than others and thus are waiting for previous ones with more work.

Idle: Load Balancing

- Some tasks may have a lot less work than others and thus are waiting for previous ones with more work.
- Thus must be flexible in choice of grain size of a task.

Outline

Executive Summary

Instruction-level Parallelism

Goal

Multiscalar Paradigm

Multiscalar Hardware

Analyzing CPU cycles

Comparison to other ILPs

Performance Evaluation

Strengths and Weaknesses

Inspired Work

Discussion

Comparison to other ILPs

- Multiscalar processor **do not have to predict every branch**, only the ones at task edges. This leads to a larger instruction window.

Comparison to other ILPs

- Multiscalar processor **do not have to predict every branch**, only the ones at task edges. This leads to a larger instruction window.
- Multiscalar processors do not have to check for conflicts when issuing loads and stores.

Comparison to other ILPs

- Multiscalar processor **do not have to predict every branch**, only the ones at task edges. This leads to a larger instruction window.
- Multiscalar processors do not have to check for conflicts when issuing loads and stores.
- Multiscalar hardware is less complex than superscalar hardware.

Outline

Executive Summary

Instruction-level Parallelism

Goal

Multiscalar Paradigm

Multiscalar Hardware

Analyzing CPU cycles

Comparison to other ILPs

Performance Evaluation

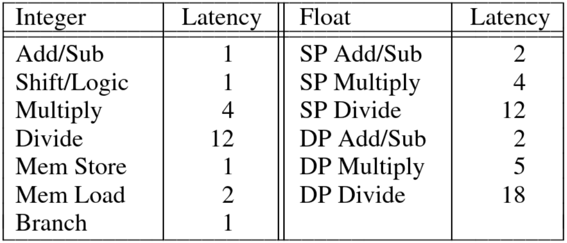
Strengths and Weaknesses

Inspired Work

Discussion

Methodology

- Simulate using MIPS instructions.

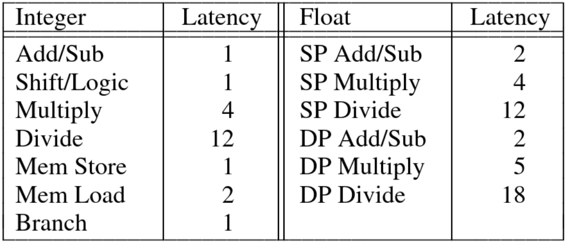


Integer	Latency	Float	Latency
Add/Sub	1	SP Add/Sub	2
Shift/Logic	1	SP Multiply	4
Multiply	4	SP Divide	12
Divide	12	DP Add/Sub	2
Mem Store	1	DP Multiply	5
Mem Load	2	DP Divide	18
Branch	1		

Figure: Functional Unit latencies

Methodology

- Simulate using MIPS instructions.
- Modified version of GCC 2.5.8 as compiler.

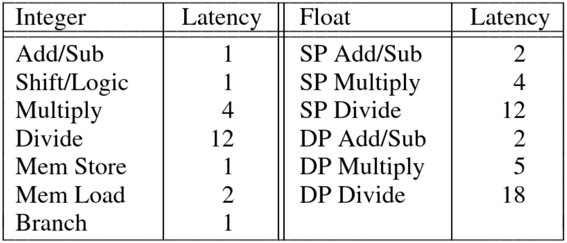


Integer	Latency	Float	Latency
Add/Sub	1	SP Add/Sub	2
Shift/Logic	1	SP Multiply	4
Multiply	4	SP Divide	12
Divide	12	DP Add/Sub	2
Mem Store	1	DP Multiply	5
Mem Load	2	DP Divide	18
Branch	1		

Figure: Functional Unit latencies

Methodology

- Simulate using MIPS instructions.
- Modified version of GCC 2.5.8 as compiler.
- 5 stage pipeline (IF, ID, EX, MEM, WB). Can be configured in-order / out-of-order and 1-way/2-way.

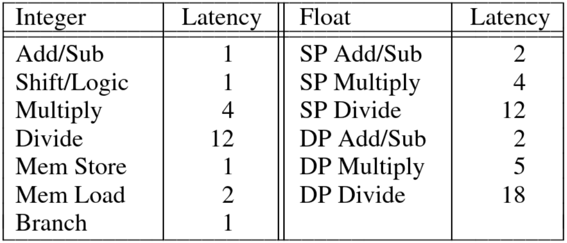


Integer	Latency	Float	Latency
Add/Sub	1	SP Add/Sub	2
Shift/Logic	1	SP Multiply	4
Multiply	4	SP Divide	12
Divide	12	DP Add/Sub	2
Mem Store	1	DP Multiply	5
Mem Load	2	DP Divide	18
Branch	1		

Figure: Functional Unit latencies

Methodology

- Simulate using MIPS instructions.
- Modified version of GCC 2.5.8 as compiler.
- 5 stage pipeline (IF, ID, EX, MEM, WB). Can be configured in-order / out-of-order and 1-way/2-way.
- 1 or 2 simple integer, 1 complex integer, 1 floating point, 1 branch and 1 memory FU.

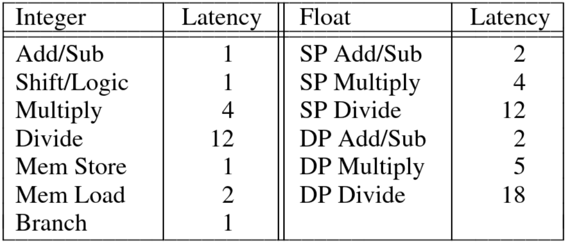


Integer	Latency	Float	Latency
Add/Sub	1	SP Add/Sub	2
Shift/Logic	1	SP Multiply	4
Multiply	4	SP Divide	12
Divide	12	DP Add/Sub	2
Mem Store	1	DP Multiply	5
Mem Load	2	DP Divide	18
Branch	1		

Figure: Functional Unit latencies

Methodology

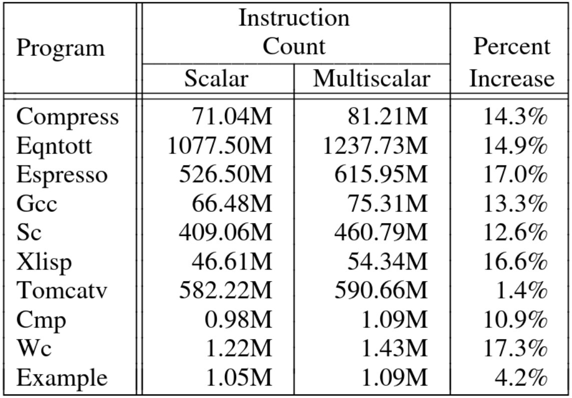
- Simulate using MIPS instructions.
- Modified version of GCC 2.5.8 as compiler.
- 5 stage pipeline (IF, ID, EX, MEM, WB). Can be configured in-order / out-of-order and 1-way/2-way.
- 1 or 2 simple integer, 1 complex integer, 1 floating point, 1 branch and 1 memory FU.
- Unidirectional ring adds 1 cycle communication latency.



Integer	Latency	Float	Latency
Add/Sub	1	SP Add/Sub	2
Shift/Logic	1	SP Multiply	4
Multiply	4	SP Divide	12
Divide	12	DP Add/Sub	2
Mem Store	1	DP Multiply	5
Mem Load	2	DP Divide	18
Branch	1		

Figure: Functional Unit latencies

Benchmarks

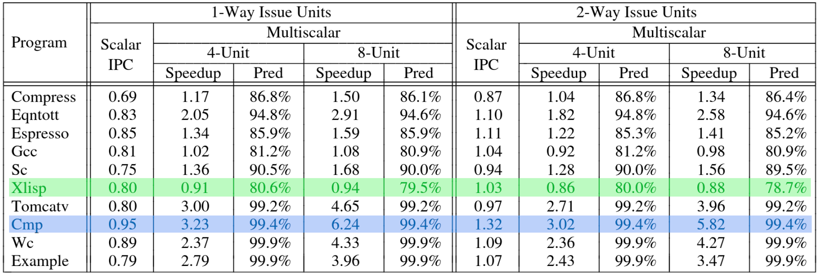


Program	Instruction Count		Percent
	Scalar	Multiscalar	Increase
Compress	71.04M	81.21M	14.3%
Eqntott	1077.50M	1237.73M	14.9%
Espresso	526.50M	615.95M	17.0%
Gcc	66.48M	75.31M	13.3%
Sc	409.06M	460.79M	12.6%
Xlisp	46.61M	54.34M	16.6%
Tomcatv	582.22M	590.66M	1.4%
Cmp	0.98M	1.09M	10.9%
Wc	1.22M	1.43M	17.3%
Example	1.05M	1.09M	4.2%

Figure: Benchmark Instruction Count

- Number of dynamic instructions listed. More in Multiscalar because of additional multiscalar instructions.

In-Order



Program	1-Way Issue Units					2-Way Issue Units				
	Scalar IPC	Multiscalar				Scalar IPC	Multiscalar			
		4-Unit		8-Unit			4-Unit		8-Unit	
		Speedup	Pred	Speedup	Pred		Speedup	Pred	Speedup	Pred
Compress	0.69	1.17	86.8%	1.50	86.1%	0.87	1.04	86.8%	1.34	86.4%
Eqntott	0.83	2.05	94.8%	2.91	94.6%	1.10	1.82	94.8%	2.58	94.6%
Espresso	0.85	1.34	85.9%	1.59	85.9%	1.11	1.22	85.3%	1.41	85.2%
Gcc	0.81	1.02	81.2%	1.08	80.9%	1.04	0.92	81.2%	0.98	80.9%
Sc	0.75	1.36	90.5%	1.68	90.0%	0.94	1.28	90.0%	1.56	89.5%
Xlisp	0.80	0.91	80.6%	0.94	79.5%	1.03	0.86	80.0%	0.88	78.7%
Tomcatv	0.80	3.00	99.2%	4.65	99.2%	0.97	2.71	99.2%	3.96	99.2%
Cmp	0.95	3.23	99.4%	6.24	99.4%	1.32	3.02	99.4%	5.82	99.4%
Wc	0.89	2.37	99.9%	4.33	99.9%	1.09	2.36	99.9%	4.27	99.9%
Example	0.79	2.79	99.9%	3.96	99.9%	1.07	2.43	99.9%	3.47	99.9%

Figure: In-Order Issue Processing Units.

In-Order

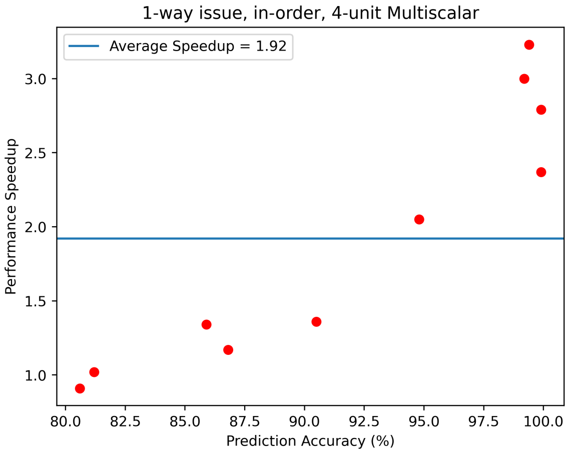


Figure: 1-way issue, in-order, 4-unit Multiscalar

In-Order

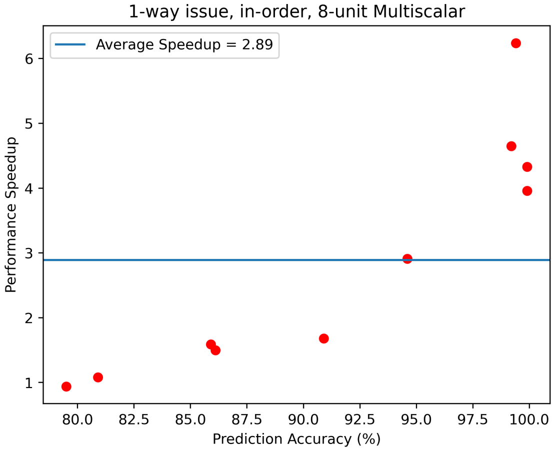


Figure: 1-way issue, in-order, 8-unit Multiscalar

In-Order

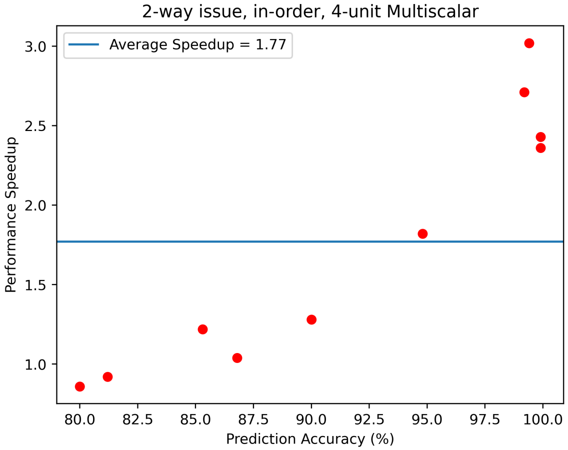


Figure: 2-way issue, in-order, 4-unit Multiscalar

In-Order

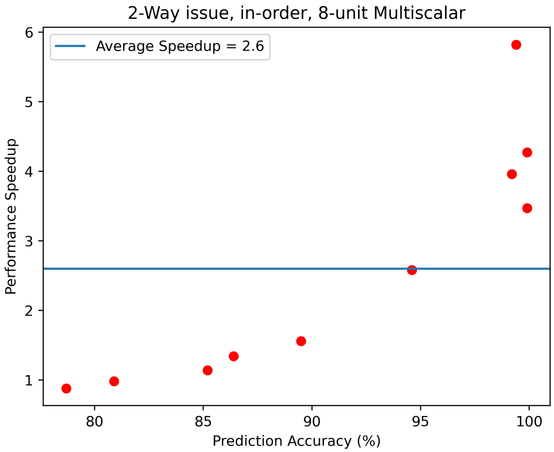
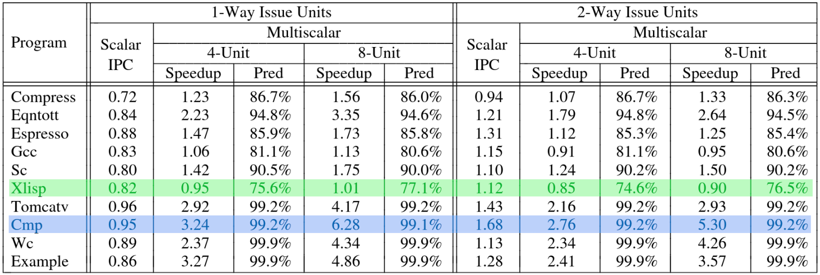


Figure: 2-way issue, in-order, 8-unit Multiscalar

Out-Order



Program	1-Way Issue Units					2-Way Issue Units				
	Scalar IPC	Multiscalar				Scalar IPC	Multiscalar			
		4-Unit		8-Unit			4-Unit		8-Unit	
		Speedup	Pred	Speedup	Pred		Speedup	Pred	Speedup	Pred
Compress	0.72	1.23	86.7%	1.56	86.0%	0.94	1.07	86.7%	1.33	86.3%
Eqntott	0.84	2.23	94.8%	3.35	94.6%	1.21	1.79	94.8%	2.64	94.5%
Espresso	0.88	1.47	85.9%	1.73	85.8%	1.31	1.12	85.3%	1.25	85.4%
Gcc	0.83	1.06	81.1%	1.13	80.6%	1.15	0.91	81.1%	0.95	80.6%
Sc	0.80	1.42	90.5%	1.75	90.0%	1.10	1.24	90.2%	1.50	90.2%
Xlisp	0.82	0.95	75.6%	1.01	77.1%	1.12	0.85	74.6%	0.90	76.5%
Tomcatv	0.96	2.92	99.2%	4.17	99.2%	1.43	2.16	99.2%	2.93	99.2%
Cmp	0.95	3.24	99.2%	6.28	99.1%	1.68	2.76	99.2%	5.30	99.2%
Wc	0.89	2.37	99.9%	4.34	99.9%	1.13	2.34	99.9%	4.26	99.9%
Example	0.86	3.27	99.9%	4.86	99.9%	1.28	2.41	99.9%	3.57	99.9%

Figure: Out-Of-Order Issue Processing Units.

Out-Order

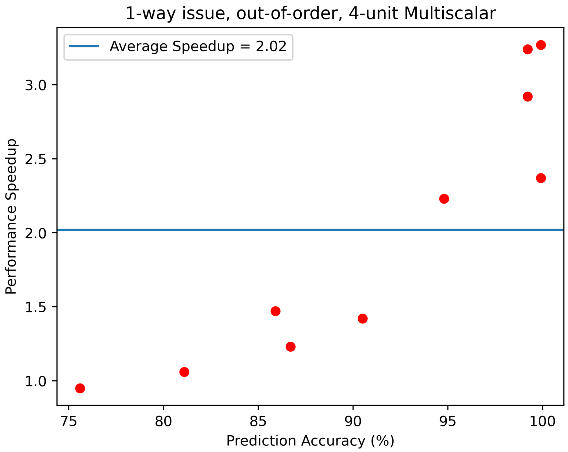


Figure: 1-way issue, out-of-order, 4-unit Multiscalar

Out-Order

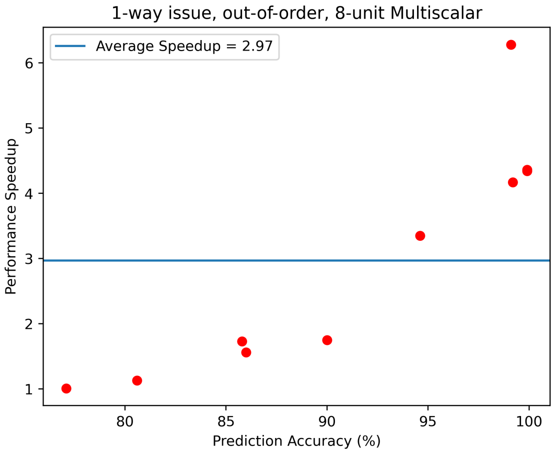


Figure: 1-way issue, out-of-order, 8-unit Multiscalar

Out-Order

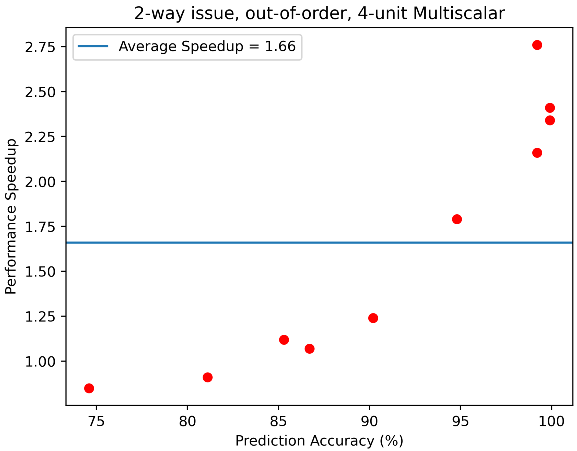


Figure: 2-way issue, out-of-order, 4-unit Multiscalar

Out-Order

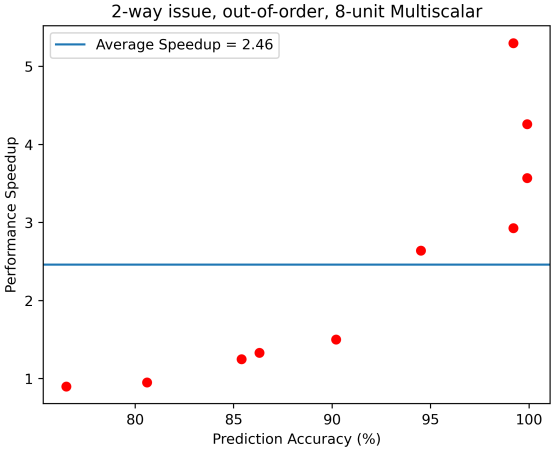


Figure: 2-way issue, out-of-order, 8-unit Multiscalar

Outline

Executive Summary

Instruction-level Parallelism

Goal

Multiscalar Paradigm

Multiscalar Hardware

Analyzing CPU cycles

Comparison to other ILPs

Performance Evaluation

Strengths and Weaknesses

Inspired Work

Discussion

Strengths

- Influential paper, large impact, enabled a lot of future research.

Strengths

- Influential paper, large impact, enabled a lot of future research.
- First group of authors to write about Multiscalar Processors. Gurindar Sohi describes the about 10 year process retrospectively here: *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers) August 1998 Pages 111–114https://doi.org/10.1145/285930.285970*

Strengths

- Influential paper, large impact, enabled a lot of future research.
- First group of authors to write about Multiscalar Processors. Gurindar Sohi describes the about 10 year process retrospectively here: *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers) August 1998 Pages 111–114https://doi.org/10.1145/285930.285970*
- Adopted by industry.

Strengths

- Influential paper, large impact, enabled a lot of future research.
- First group of authors to write about Multiscalar Processors. Gurindar Sohi describes the about 10 year process retrospectively here: *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers) August 1998 Pages 111–114https://doi.org/10.1145/285930.285970*
- Adopted by industry.
- Authors released updates on `http://pages.cs.wisc.edu/~mscalar/` (last edited 2008) and followed if they were still investigating a specific subject of the paper or were vague: Dead register analysis, Control flow speculation.

Strengths

- Influential paper, large impact, enabled a lot of future research.
- First group of authors to write about Multiscalar Processors. Gurindar Sohi describes the about 10 year process retrospectively here: *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers) August 1998 Pages 111–114https://doi.org/10.1145/285930.285970*
- Adopted by industry.
- Authors released updates on `http://pages.cs.wisc.edu/~mscalar/` (last edited 2008) and followed if they were still investigating a specific subject of the paper or were vague: Dead register analysis, Control flow speculation.
- With high prediction accuracy come large speedups.

Strengths

- Influential paper, large impact, enabled a lot of future research.
- First group of authors to write about Multiscalar Processors. Gurindar Sohi describes the about 10 year process retrospectively here: *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers) August 1998 Pages 111–114https://doi.org/10.1145/285930.285970*
- Adopted by industry.
- Authors released updates on `http://pages.cs.wisc.edu/~mscalar/` (last edited 2008) and followed if they were still investigating a specific subject of the paper or were vague: Dead register analysis, Control flow speculation.
- With high prediction accuracy come large speedups.
- Well explained examples were provided.

Weaknesses

- Seems to be heavily reliant on code rewriting / specific compiler.

Weaknesses

- Seems to be heavily reliant on code rewriting / specific compiler.
- Only hints on how tasks are to be split up were given.

Weaknesses

- Seems to be heavily reliant on code rewriting / specific compiler.
- Only hints on how tasks are to be split up were given.
- ISA needs to be (minimally) changed.

Weaknesses

- Seems to be heavily reliant on code rewriting / specific compiler.
- Only hints on how tasks are to be split up were given.
- ISA needs to be (minimally) changed.
- Speedup does not appear to be capped at 8-units for highly predictable tasks, would have been interesting to see how the speedup behaves with 16-units on highly predictable executions.

Outline

Executive Summary

Instruction-level Parallelism

Goal

Multiscalar Paradigm

Multiscalar Hardware

Analyzing CPU cycles

Comparison to other ILPs

Performance Evaluation

Strengths and Weaknesses

Inspired Work

Discussion

Inspired Work: Slipstream processors

- **Slipstream processors:** Create shorter but equivalent by removing ineffectual computation and computation related to highly-predictable control flow. Concurrently run original and short program. Shorter program speculatively runs ahead of original program and supplies original program with control and data flow outcomes. The full program then uses that information to execute more efficiently and validates the speculative, shorer program.

Inspired Work: Slipstream processors

- **Slipstream processors:** Create shorter but equivalent by removing ineffectual computation and computation related to highly-predictable control flow. Concurrently run original and short program. Shorter program speculatively runs ahead of original program and supplies original program with control and data flow outcomes. The full program then uses that information to execute more efficiently and validates the speculative, shorter program.
- *Purser, Zach, Karthik Sundaramoorthy, and Eric Rotenberg. "A study of slipstream processors." Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture. 2000.*

Inspired Work: Thread level speculation

- **Thread level speculation:** Speculatively execute portions of code parallel to the main thread in an independent thread. May need to make assumptions about input values. If assumptions are violated the speculative thread must be discarded and squashed.

Inspired Work: Thread level speculation

- **Thread level speculation:** Speculatively execute portions of code parallel to the main thread in an independent thread. May need to make assumptions about input values. If assumptions are violated the speculative thread must be discarded and squashed.
- *Steffan, J. Greggory, et al. ”A scalable approach to thread-level speculation.” ACM SIGARCH Computer Architecture News 28.2 (2000): 1-12.*

Outline

Executive Summary

Instruction-level Parallelism

Goal

Multiscalar Paradigm

Multiscalar Hardware

Analyzing CPU cycles

Comparison to other ILPs

Performance Evaluation

Strengths and Weaknesses

Inspired Work

Discussion

Discussion

Optimizing Forwarding of Values

What can be done to potentially reduce the number of values which have to be forwarded?

Optimizing Forwarding of Values

- Use **Liveness analysis / dead value analysis** or similar to determine if values are needed later on. If not, there is no need to forward. (Mentioned in paper)

Optimizing Forwarding of Values

- Use **Liveness analysis / dead value analysis** or similar to determine if values are needed later on. If not, there is no need to forward. (Mentioned in paper)
- Split tasks such that local variables local to some code are fully incorporated in that task.

Optimizing Forwarding of Values

- Use **Liveness analysis / dead value analysis** or similar to determine if values are needed later on. If not, there is no need to forward. (Mentioned in paper)
- Split tasks such that local variables local to some code are fully incorporated in that task.
- *Exploiting Dead Value Information Milo M. Martin, Amir Roth, Charles N. Fischer 30th Annual international Symposium on Microarchitecture (MICRO-30), Dec 1997.*

Microarchitecture changes

How can the microarchitecture be changed to improve certain metrics like space and latency?

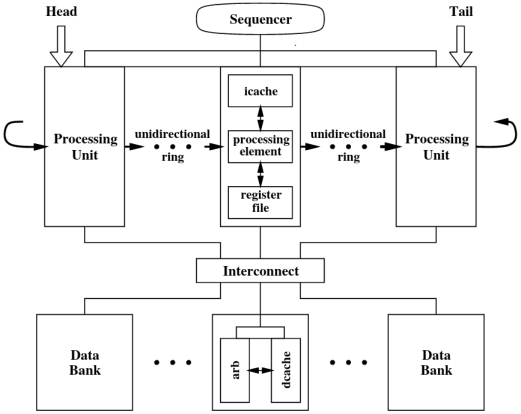


Figure: Example Hardware

Microarchitecture changes

- **Space**: Processing units share functional units like floating point units. Negative: May have to wait for other processing units to finish using parts of hardware.

Microarchitecture changes

- **Space**: Processing units share functional units like floating point units. Negative: May have to wait for other processing units to finish using parts of hardware.
- **Latency**: Move data bank directly next to processing units. Negative: Need to handle inconsistent caches and buffers and forward information.

Can we reuse information from squashed tasks?

Can we reuse information from squashed tasks?

- Some parts of a task may not depend on previous values but may still be squashed.

Can we reuse information from squashed tasks?

- Some parts of a task may not depend on previous values but may still be squashed.
- Could keep a record of that information.

Can we reuse information from squashed tasks?

- Some parts of a task may not depend on previous values but may still be squashed.
- Could keep a record of that information.
- Could split tasks such that there are less dependencies and with that one would probably have to increase the amount of units.

Can we reuse information from squashed tasks?

- Some parts of a task may not depend on previous values but may still be squashed.
- Could keep a record of that information.
- Could split tasks such that there are less dependencies and with that one would probably have to increase the amount of units.
- *Register Integration: A Simple and Efficient Implementation of Squash Reuse Amir Roth and Gurindar S. Sohi 33rd International Symposium on Microarchitecture (MICRO-33), Dec. 10-13, 2000.*

Task selection

- Which heuristics could be helpful in determining the task boundaries?

Task selection

- Which heuristics could be helpful in determining the task boundaries?
- Should we reorder instructions and how?

Task selection

- Use heuristics:

Task selection

- Use heuristics:
- **Task size**

Task selection

- Use heuristics:
- **Task size**
- **Control Flow**

Task selection

- Use heuristics:
- **Task size**
- **Control Flow**
- **Data Dependence**

Task selection

- Use heuristics:
- **Task size**
- **Control Flow**
- **Data Dependence**
- *T. N. Vijaykumar and G. S. Sohi, "Task selection for a multiscalar processor," Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture, 1998, pp. 81-92, doi: 10.1109/MICRO.1998.742771.*