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

Goal: Execute many instructions in parallel per cycle.
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.
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
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 into 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 into 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 into 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.
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:
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
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**.
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

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

```c
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
Example Code Segment

```c
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

```c
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 == LLEE(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

- Multiscalr Programs
- Sequencer
Multiscalar Paradigm; Next Steps

- Multiscalar Programs
- Sequencer
- Communication between tasks
Multiscalar Paradigm; Next Steps

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

Controlflow information can be statically determined and is placed in a task descriptor. The task descriptor may be placed within program text or in a single location.
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.
Figure: Example Program

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

<table>
<thead>
<tr>
<th>Targ Spec</th>
<th>Branch, Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Targ1</td>
<td>OUTER</td>
</tr>
<tr>
<td>Targ2</td>
<td>OUTERFALLOUT</td>
</tr>
<tr>
<td>Create mask</td>
<td>$4,$8,$17,$20,$23</td>
</tr>
</tbody>
</table>

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
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.

**Figure:** Example Program
Example Program

<table>
<thead>
<tr>
<th>Targ Spec</th>
<th>Branch, Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>Targ1</td>
<td>OUTER</td>
</tr>
<tr>
<td>Targ2</td>
<td>OUTERFALLOUT</td>
</tr>
<tr>
<td>Create mask</td>
<td>$4, $8, $17, $20, $23</td>
</tr>
</tbody>
</table>

**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:**

- 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.

**Figure:** Example Program
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

Key Components:
- Sequencer
- Processing Units

Figure: Example Hardware
Multiscalar Hardware; One of many implementations

Key Components:

- Sequencer
- Processing Units
- Data Banks

Figure: Example Hardware
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.
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.
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 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.

<table>
<thead>
<tr>
<th>Integer</th>
<th>Latency</th>
<th>Float</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/Sub</td>
<td>1</td>
<td>SP Add/Sub</td>
<td>2</td>
</tr>
<tr>
<td>Shift/Logic</td>
<td>1</td>
<td>SP Multiply</td>
<td>4</td>
</tr>
<tr>
<td>Multiply</td>
<td>4</td>
<td>SP Divide</td>
<td>12</td>
</tr>
<tr>
<td>Divide</td>
<td>12</td>
<td>DP Add/Sub</td>
<td>2</td>
</tr>
<tr>
<td>Mem Store</td>
<td>1</td>
<td>DP Multiply</td>
<td>5</td>
</tr>
<tr>
<td>Mem Load</td>
<td>2</td>
<td>DP Divide</td>
<td>18</td>
</tr>
<tr>
<td>Branch</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure:** Functional Unit latencies
Methodology

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

<table>
<thead>
<tr>
<th>Integer</th>
<th>Latency</th>
<th>Float</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/Sub</td>
<td>1</td>
<td>SP Add/Sub</td>
<td>2</td>
</tr>
<tr>
<td>Shift/Logic</td>
<td>1</td>
<td>SP Multiply</td>
<td>4</td>
</tr>
<tr>
<td>Multiply</td>
<td>4</td>
<td>SP Divide</td>
<td>12</td>
</tr>
<tr>
<td>Divide</td>
<td>12</td>
<td>DP Add/Sub</td>
<td>2</td>
</tr>
<tr>
<td>Mem Store</td>
<td>1</td>
<td>DP Multiply</td>
<td>5</td>
</tr>
<tr>
<td>Mem Load</td>
<td>2</td>
<td>DP Divide</td>
<td>18</td>
</tr>
<tr>
<td>Branch</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Integer</th>
<th>Latency</th>
<th>Float</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/Sub</td>
<td>1</td>
<td>SP Add/Sub</td>
<td>2</td>
</tr>
<tr>
<td>Shift/Logic</td>
<td>1</td>
<td>SP Multiply</td>
<td>4</td>
</tr>
<tr>
<td>Multiply</td>
<td>4</td>
<td>SP Divide</td>
<td>12</td>
</tr>
<tr>
<td>Divide</td>
<td>12</td>
<td>DP Add/Sub</td>
<td>2</td>
</tr>
<tr>
<td>Mem Store</td>
<td>1</td>
<td>DP Multiply</td>
<td>5</td>
</tr>
<tr>
<td>Mem Load</td>
<td>2</td>
<td>DP Divide</td>
<td>18</td>
</tr>
<tr>
<td>Branch</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Integer</th>
<th>Latency</th>
<th>Float</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/Sub</td>
<td>1</td>
<td>SP Add/Sub</td>
<td>2</td>
</tr>
<tr>
<td>Shift/Logic</td>
<td>1</td>
<td>SP Multiply</td>
<td>4</td>
</tr>
<tr>
<td>Multiply</td>
<td>4</td>
<td>SP Divide</td>
<td>12</td>
</tr>
<tr>
<td>Divide</td>
<td>12</td>
<td>DP Add/Sub</td>
<td>2</td>
</tr>
<tr>
<td>Mem Store</td>
<td>1</td>
<td>DP Multiply</td>
<td>5</td>
</tr>
<tr>
<td>Mem Load</td>
<td>2</td>
<td>DP Divide</td>
<td>18</td>
</tr>
<tr>
<td>Branch</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*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.

<table>
<thead>
<tr>
<th>Integer</th>
<th>Latency</th>
<th>Float</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/Sub</td>
<td>1</td>
<td>SP Add/Sub</td>
<td>2</td>
</tr>
<tr>
<td>Shift/Logic</td>
<td>1</td>
<td>SP Multiply</td>
<td>4</td>
</tr>
<tr>
<td>Multiply</td>
<td>4</td>
<td>SP Divide</td>
<td>12</td>
</tr>
<tr>
<td>Divide</td>
<td>12</td>
<td>DP Add/Sub</td>
<td>2</td>
</tr>
<tr>
<td>Mem Store</td>
<td>1</td>
<td>DP Multiply</td>
<td>5</td>
</tr>
<tr>
<td>Mem Load</td>
<td>2</td>
<td>DP Divide</td>
<td>18</td>
</tr>
<tr>
<td>Branch</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure:** Functional Unit latencies
## Benchmarks

<table>
<thead>
<tr>
<th>Program</th>
<th>Instruction Count</th>
<th>Percent Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Scalar</td>
<td>Multiscalar</td>
</tr>
<tr>
<td>Compress</td>
<td>71.04M</td>
<td>81.21M</td>
</tr>
<tr>
<td>Eqntott</td>
<td>1077.50M</td>
<td>1237.73M</td>
</tr>
<tr>
<td>Espresso</td>
<td>526.50M</td>
<td>615.95M</td>
</tr>
<tr>
<td>Gcc</td>
<td>66.48M</td>
<td>75.31M</td>
</tr>
<tr>
<td>Sc</td>
<td>409.06M</td>
<td>460.79M</td>
</tr>
<tr>
<td>Xlisp</td>
<td>46.61M</td>
<td>54.34M</td>
</tr>
<tr>
<td>Tomcatv</td>
<td>582.22M</td>
<td>590.66M</td>
</tr>
<tr>
<td>Cmp</td>
<td>0.98M</td>
<td>1.09M</td>
</tr>
<tr>
<td>Wc</td>
<td>1.22M</td>
<td>1.43M</td>
</tr>
<tr>
<td>Example</td>
<td>1.05M</td>
<td>1.09M</td>
</tr>
</tbody>
</table>

**Figure:** Benchmark Instruction Count

- Number of dynamic instructions listed. More in Multiscalar because of additional multiscalar instructions.
### In-Order Issue Processing Units

<table>
<thead>
<tr>
<th>Program</th>
<th>1-Way Issue Units</th>
<th>2-Way Issue Units</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Scalar IPC</td>
<td>Multiscalar</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>Pred</td>
</tr>
<tr>
<td>Compress</td>
<td>0.69</td>
<td>1.17</td>
</tr>
<tr>
<td>Eqntott</td>
<td>0.83</td>
<td>2.05</td>
</tr>
<tr>
<td>Espresso</td>
<td>0.85</td>
<td>1.34</td>
</tr>
<tr>
<td>Gcc</td>
<td>0.81</td>
<td>1.02</td>
</tr>
<tr>
<td>Sc</td>
<td>0.75</td>
<td>1.36</td>
</tr>
<tr>
<td>Xlisp</td>
<td>0.80</td>
<td>0.91</td>
</tr>
<tr>
<td>Tomcatv</td>
<td>0.80</td>
<td>3.00</td>
</tr>
<tr>
<td>Cmp</td>
<td>0.95</td>
<td>3.23</td>
</tr>
<tr>
<td>Wc</td>
<td>0.89</td>
<td>2.37</td>
</tr>
<tr>
<td>Example</td>
<td>0.79</td>
<td>2.79</td>
</tr>
</tbody>
</table>

**Figure:** In-Order Issue Processing Units.
Performance Evaluation

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

<table>
<thead>
<tr>
<th>Program</th>
<th>1-Way Issue Units</th>
<th>2-Way Issue Units</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Scalar IPC</td>
<td>Multiscalar</td>
</tr>
<tr>
<td></td>
<td>4-Unit</td>
<td>8-Unit</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>Pred</td>
</tr>
<tr>
<td>Compress</td>
<td>0.72</td>
<td>1.23</td>
</tr>
<tr>
<td>Eqntott</td>
<td>0.84</td>
<td>2.23</td>
</tr>
<tr>
<td>Espresso</td>
<td>0.88</td>
<td>1.47</td>
</tr>
<tr>
<td>Gcc</td>
<td>0.83</td>
<td>1.06</td>
</tr>
<tr>
<td>Sc</td>
<td>0.80</td>
<td>1.42</td>
</tr>
<tr>
<td>Xlisp</td>
<td>0.82</td>
<td>0.95</td>
</tr>
<tr>
<td>Tomcatv</td>
<td>0.96</td>
<td>2.92</td>
</tr>
<tr>
<td>Cmp</td>
<td>0.95</td>
<td>3.24</td>
</tr>
<tr>
<td>Wc</td>
<td>0.89</td>
<td>2.37</td>
</tr>
<tr>
<td>Example</td>
<td>0.86</td>
<td>3.27</td>
</tr>
</tbody>
</table>

**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

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

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

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

Average Speedup = 2.46
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 111–114 https://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 111–114 [https://doi.org/10.1145/285930.285970](https://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 111–114 https://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 111–114 [https://doi.org/10.1145/285930.285970](https://doi.org/10.1145/285930.285970)
- Adopted by industry.
- Authors released updates on [http://pages.cs.wisc.edu/~mscalar/](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 and Weaknesses

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 111–114 [https://doi.org/10.1145/285930.285970](https://doi.org/10.1145/285930.285970)
- Adopted by industry.
- Authors released updates on [http://pages.cs.wisc.edu/~mscalar/](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.

▶ 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.
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, shorer program.

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.

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