Multiscalar Processors

Executive Summary

• **Problem**: Improve the performance of sequential execution

• **Goal**: Execute as many instructions as possible per cycle (IPC)

• **Key idea**: Improve the instruction-level parallelism (ILP) through the “Multiscalar Paradigm“, which divides the program into a collection of tasks to increase ILP

• **Mechanism**: Each task is assigned to one of many processing elements (PE) at runtime for independent execution

• **Result**: Multiscalar Processors significantly improves ILP in parallelisable workloads
Agenda

• Basic Concepts
• Multiscalar Paradigm
• Design Analysis
• Evaluation
• Strengths and Weaknesses
• Discussion
Basic Concepts
Instruction-level Parallelism

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

- Throughput still 1 instruction per cycle

5-stage single-issue in-order pipeline
Basic Concepts
Instruction-level Parallelism

• **Superscalar**: Fetch and dispatch multiple instructions at once

• Can be of any implementation, so long as throughput > 1 instruction/cycle

• Use a **single PC** to step through program and establish window of operations

5-stage *dual-issue* in-order pipeline
Basic Concepts
Instruction-level Parallelism

• **Out-of-order** (OoO): Instructions execute in any order that does not violate data dependencies

• Coupled with multi-issue for superscalar throughput

• „Order“ still mandated by memory model (consistency)

6-port OoO machine with 3 ALU groups
Basic Concepts
Instruction-level Parallelism

• **Dataflow Execution Model:** Instruction execute once input is available
  • No program counter!
  • Mostly used in DSP, HLS, etc.
  • Difficult to build general purpose processor

Simplified vector dot product
Observation of Previous Mechanisms

• Most instructions are *independent* on each other
  
  • Thus, a sequential execution plan does not exploit such independence in terms of improving parallelism

• **Key constraint** in previous methods: *stall* instructions until previous control and data dependencies are resolved
Multiscalar Paradigm
Agenda
Multiscalar Paradigm

- Philosophy and Concepts
  - Multiscalar Program
  - Multiscalar Hardware
Philosophy and Concepts
Multiscalar Paradigm

• **Cooperation** between *software* and *hardware*:
  • Split the programs into *tasks* using the control flow graph (CFG)
  • Speculatively distribute tasks into parallel processing elements (PE)
• Use **separate PCs** to sequence through the program
  • Resemble sequential *appearance* by constraining dispatch and commit
Control Flow Graph (CFG)

Important Concepts

• Directed graph that shows the control flow of a program
  • **Basic blocks** (nodes) and **control flow** (edges)
• First instruction is the **unique** entry point of basic block
• Last instruction is the **only** control flow instruction (jump, etc.)
Tasks

Important Concepts

- A task is a portion of the entire CFG
- A **contiguous** region of a **dynamic** instruction sequence
- Assigned to PEs for execution
- Possibly **dependent** on each other

Example of a program with 3 tasks
Imposing Sequential Appearance
Philosophy and Concepts

• Challenge when instruction per cycle > 1:
  • Ensure the PEs adhere to **sequential** execution **semantics**

• Important definitions to maintain a sequential semantics:
  • Order between PEs
  • Speculative task execution
Order Between PEs
Imposing Sequential Appearance

• Task-level sequential order: circular queue of PEs

• Tasks consume and produce values bound to registers or memory

• Maintain one single set of registers and memory locations
• Task-level sequential order: circular queue of PEs

• Tasks consume and produce values bound to registers or memory

• Maintain one single set of registers and memory locations
Order Between PEs

Imposing Sequential Appearance

- Task-level sequential order: **circular queue** of PEs

- Tasks **consume and produce** values bound to **registers** or **memory**

- Maintain **one single set** of registers and memory locations
Speculative Task Execution

Imposing Sequential Appearance

- Speculative tasks due to:
  - **Control** speculation i.e. branch prediction
  - **Data** speculation i.e. non-conformant to memory model
- Resolve data dependency violations with **address resolution buffers**
  - In case of unresolvable **conflict**, successors must be **squashed**
- Tasks are **retired in the order** as they are added
Agenda
Multiscalar Paradigm

• Philosophy and Concepts
• Multiscalar Program
• Multiscalar Hardware
Example Program
Multiscalar Program

- Linear search through linked list for symbol
- Outer loop iterates through a list of symbols searched
- Inner loop iterates through the linked list of items

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

    // 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 tail
    if (!list)
        addlist(symbol);
}
```
Multiscalar Programs
Multiscalar Paradigm

• Should enable **fast walk** through the CFG to distribute tasks on many PEs

• The **sequencer** chooses one possible successors of tasks (statically determined) to continue walk

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

    // 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 tail
    if (!list)
        addlist(symbol);
}
```
Multiscalar Programs

Multiscalar Paradigm

• Should enable fast walk through the CFG to distribute tasks on many PEs

• The sequencer chooses one possible successors of tasks (statically determined) to continue walk
Multiscalar Programs
Multiscalar Paradigm

• Communication between tasks implemented as minimal ISA changes

• **Forward** „live-out“ values to successor tasks

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

    // 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 tail
if (!list) {
    addlist(symbol);
}
```
Multiscalar Programs

Multiscalar Paradigm

• Communication between tasks implemented as minimal ISA changes

• **Forward „live-out“ values** to successor tasks
Agenda
Multiscalar Paradigm

• Philosophy and Concepts
• Multiscalar Program
• Multiscalar Hardware
Multiscalar Hardware
Multiscalar Paradigm

- Key components:
  - **Sequencer**
  - **Processing Element**
  - **Address Resolution Buffer**
  - Otherwise a traditional SoC
Multiscalar Hardware
Multiscalar Paradigm

- **Sequencer:**
  - Determine *order* of tasks
  - Fetch descriptor and executes task
  - **Predict** next task

An example of a 3-unit multiscalar processor
Multiscalar Hardware
Multiscalar Paradigm

• **Processing Element:**
  - **Independently** fetch and execute tasks
  - Forward register values through the uni-directional ring

An example of a 3-unit multiscalar processor
Multiscalar Hardware

Multiscalar Paradigm

- **Address Resolution Buffer:**
  - Holds *speculative* memory operations before retirement
  - Detects and corrects *dependency violations*

An example of a 3-unit multiscalar processor
Design Analysis
Design Analysis
Breakdown of (useless) CPU cycles

• Objective: all PEs should perform useful computation at all times

• Try to avoid:
  • **Squashing** due to *wrong speculation*
  • **Stalling** due to *waiting for values*
  • **Idle** due to *no schedulable task*
Solutions
Avoiding Useless CPU Cycles

• Observation: incorrect branch prediction cause squashing
  • => early validation of prediction; optimise code structure
• Observation: inter-task dependencies serialise execution
  • => source-level change to reduce dependencies
• Observation: short tasks wait for very long tasks
  • Perform load-balancing by adjusting granularity of tasks
Design Analysis
Comparison with other ILP Paradigms

• Does **not** require **branch prediction** on every branch
  • Only on task edges! => larger instruction window

• Does **not** have to check load-store **conflicts** on issuance
  • Conflict checked by ARBs instead

• **Less complex** hardware
  • Simple, in-order cores sufficient to achieve high IPC
Evaluation
Evaluation
Experiment Setup

• Big-endian **MIPS simulator** with cycle models
  • 5-stage pipeline
  • Configurable in/out-of-order & single/dual issue
• Modified GCC 2.5.8 compiler for multiscalar programs
• Benchmarks from SPECfp92 and GNU Coreutils
Evaluation
Increase in Dynamic Instruction Count cf. Scalar

More instruction in multiscalar: Additional multiscalar data instructions (release)
Evaluation
Speedup cf. Scalar and Prediction Accuracy

Prediction accuracy
Speedup=1
Speedup=4
Speedup=8

In-order 4-unit
Out-of-order 4-unit
In-order 8-unit
Out-of-order 8-unit

Compress  Eqntott  Espresso  Gcc  Sc  Xlisp  Tomcatv  Cmp  Wc  Example

Speedup < 1!
Speedup = ~80% theoretical
Evaluation

Speedup vs. Prediction Accuracy

Speedup largely correlates to task prediction accuracy
Linear speedup in suitable workloads
Strengths and Weaknesses
Strengths

- Completely new (in 1995) compared to other ILP paradigms
  - **Speculative** control and **data** flow
  - **Cooperation** between compiler and hardware
- Influential with large impact: 1339 citations since publication
  - Address Resolution Buffer: used by **virtually all processors** nowadays (known as ROB)
- Better known as **speculative thread parallelism**
  - Implemented in IBM XL C/C++ (-qsmp=speculative), OpenMP TLS extension
Weaknesses

• Important details *glossed over*: how to implement the *sequencer*, *ARB*, etc.?
  • Time-proved difficult hardware design: *no full, real hardware ever*

• **Dependent** on compiler optimisations: story of Itanium
  • Bad compiler (in 1995!) => poor performance

• Less vigorous evaluation (in today’s standards)
  • Missing comparison with other ILP paradigms
  • No cycle-accurate simulation model / synthesizable implementation
Acknowledgements

- **Special thanks** to mentors of this presentation
  - Haocong Luo
  - Rahul Bera
  - Nisa Bostancı
- Thanks to the teaching team for picking the papers and teaching the basics
- …and the audience for listening! Please engage in the discussion :)}
Discussion
Transferability into Other Fields

Discussion

• Thread Level Speculation (TLS):
  • **Speculatively** execute a section of computer code in a separate independent thread

• Software on **SMT processors** v.s. hardware-based approach?

• Task creation? Passing values? Rollback?

A Survey on Thread-Level Speculation Techniques

ALVARO ESTEBANEZ, DIEGO R. LLANOS, and ARTURO GONZALEZ-ESCRIBANO, Universidad de Valladolid
Discussion

• Multiscalar execution model:
  • CFG slicing, independent runtime task scheduling

• Explicit Data Graph Execution (EDGE): Microsoft & DARPA
  • „Dataflow execution“: execute hyperblocks on all PEs at the same time

• Difference?
Hardware-Software Co-design

Discussion

- Open ended: compiler-assisted hints for higher hardware performance
  - Multiscalar processors: task division, "stop" and "forward" bits
  - MIPS & SPARC: delay slots
- More?
Thanks for your attention
Backup Slides
Basic Concepts
Instruction-level Parallelism

• **Very Long Instruction Word (VLIW):** Encode multiple "instructions" in one instruction

• Compiler statically schedule instructions to be executed in parallel

4-way VLIW with 16-byte instructions
Register and Memory Synchronisation

Order between PEs

Registers

• Produce & consume can be **statically** determined

• Produced: **forward** to successor

• Consumed: **wait** for value

Memory

• **Known** locations: same as registers

• Unknown location:
  
  • **Aggressively speculatively** load (cf. conservatively wait)
Augmenting Binaries
Multiscalar Programs

• **Support** executing old binaries, albeit *slowly* due to lack of parallelism

• For *existing* binaries:
  • Generate CFG and task structure and add to binary

• Possible for migrating existing, non-multiscalar binaries to multiscalar