Scheduling for VLIW Processors

Giovanni Agosta
agosta@elet.polimi.it

Politecnico di Milano
Overview

The Nature of Parallelism
  • Instruction–Level Parallelism
  • Loop–Level Parallelism
  • Task–Level Parallelism

VLIW Processors

Instruction Scheduling
  • Trace Scheduling
    • Exercise
  • Superblock Scheduling
  • Modulo Scheduling
The Nature of Parallelism

- Concurrent execution of multiple instructions
  - Parallelism is a property which allows the simultaneous execution of two or more instructions, or their execution out of the sequential order imposed by the programmer
  - Parallelised instructions may stem from the same control flow or from two different flows
The Nature of Parallelism

- Preservation of program semantics
  - The operation of parallelising a program must preserve its semantics, so parallelisation of instructions which read or write the same variable is forbidden
  - The parallelising compilers must check data dependencies among the instructions
Granularity of Parallelism

- **Task level parallelism (coarse grain)**
  - Parallelised blocks are processes
  - Available on multiprocessors

- **Loop level parallelism (medium grain)**
  - Parallelised blocks are loop iterations
  - Available on multiprocessors through program transformations

- **Instruction level parallelism (fine grain)**
  - Parallelised blocks are instructions
  - Available on Superscalars and VLIWs
Instruction-Level Parallelism

- Fine-grain parallelism
  - ILP works on small blocks so
  - It makes it easier to exploit small amounts of parallelism
  - Exploits parallelism which is common in most programs, even those traditionally not considered parallel-intensive
  - ILP cannot work through control flow barriers
  - It works only within the boundaries of a basic block
Instruction–Level Parallelism

- ILP & Speculative execution
  Speculation can overcome constraints due to if-then-else constructs

- ILP & Loops
  Exploiting ILP parallelism through loop barriers allows for a more aggressive parallelisation than LLP
  However, this is not bound to always bring advantages
Loop–Level Parallelism

- Medium-grain parallelism
  - LLP works on loops, where independent iterations may be parallelised
  - LLP can be exploited by most machine architectures, though it may need special transformations
  - On VLIW machines, Modulo Scheduling is needed for LLP exploitation
  - On multiprocessors, various optimizations and scheduling techniques are employed
Loop–Level Parallelism

- LLP & Program transformations
  - These are needed to extract the parallel schemas from generic loops
  - Loop skewing is a common transformation which restructures the iteration space
  - Many of these transformations allow for a common representation
    - Affine transforms, Unimodular transforms, etc.
Task-Level Parallelism

- Coarse-grain parallelism
  - TLP works on processes, function, or otherwise large chunks of code, where parallelism may be achieved with low overhead costs
  - TLP is usually exploited by multiprocessors, since they allow for maximum flexibility
Task–Level Parallelism

- TLP & User intervention
  - TLP often requires user intervention to be exploited, forcing him to plan the software development with special constraints
  - Often, processes must be declared as concurrent, or special language constructs must be used to mark parallelisable regions of a program
Hardware Scheduling

Pros:

- Some dependencies are only known at runtime
- Makes life easier for compiler writers
- Code is portable between machines with same ISA and different levels of parallelism

Cons:

- More hardware complexity
- Less time to perform dependence analysis
Hardware Scheduling Techniques

- Out-of-order execution:
  - An instruction would stall the pipeline
  - Subsequent instructions may be issued if no dependencies are violated

- Techniques:
  - Scoreboarding
  - Tomasulo’s Algorithm
Scoreboarding

- Pipeline similar to DLX, with multiple functional units
- Split ID stage
  1. Instruction Decode: in-order
  2. Read Operands: out-of-order
- Out-of-Order completion may generate WAR and WAW hazards (solved with stalls)
  1. WAR are solved by reading operands only in the Read Operand phase
  2. WAW are solved by stalling the pipeline, and completing in-order.
Tomasulo’s Algorithm

Solves WAR and WAW hazards with a more advanced technique:

Reservation Stations (RS)
1. Buffer operands while the instruction is stalled (removes WAR)
2. Redirects operand read from registers to RS (solves WAW)

Reservation Stations store:
1. Instruction opcode
2. Instruction operands or pointers to RS holding the instructions that will compute them
VLIW processors

- Very Long Instruction Word Architectures
- Oriented to the exploitation of Instruction–Level Parallelism
- Low hardware complexity
- Explicit parallelism
- Single control flow
VLIW processors

- HW dependence checking and dynamic scheduling (as in superscalars) have negative impact:
  - On the chip area (more logic packed into the issue stage)
  - On power consumption (more circuits dissipates more power)

- How to reduce the amount of hardware?
  - Move as many decisions as possible from runtime to compile time
  - Scheduling is a good candidate
VLIW processors

cyc  Scalar  Very Long Instruction Word
1   add r1=r2,r3 | add r1=r2,r3  sub r4=r4,1  mul r5=r2,r3
2   sub r4=r4,1  | mov r3=r1   -     -
3   mul r5=r2,r3
4   mov r3=r1

- No instruction reordering performed by the hardware
- Parallel execution is provided for by the compiler (scheduling)
- Instructions (syllables) are grouped into VLIWs (bundles)
VLIW processors

Non-unit latencies
- Operation are not seen as atomic by the compiler.
- The compiler must insert explicit NOPs after any operation that last more than 1 cycle.

```
1   add r1=r2,r3   sub r4=r4,1   mul r5=r2,r3   lw r5, 0[r7]
2   mov r3=r1      add r1=r1,r4   NOP     NOP
```

- Even if there were additional instructions with no dependencies, it would be impossible to schedule them in bundle 2.
VLIW processors

- Dependency checking is performed by the compiler.
- The hardware executes operations regardless of data flow hazards.
- NOPs are inserted by the compiler to solve RAW hazards.
- Register allocation is used to solve false dependencies.
- Compile-time scheduling helps in removing WAR and WAW hazards.
- Control hazards are solved by the hardware.
VLIW Advantages

- There is much more time for scheduling at compile time
- Non-local scheduling can be employed
- Computationally costly algorithms can be applied by the compiler
- These algorithms allow to
  - Perform near-optimal local scheduling
  - Perform scheduling of instructions from different regions of the code
  - Schedule entire loops
VLIW Disadvantages

- Limited or non-existent code compatibility
  - Even machines with the same ISA (at syllable level) are usually incompatible at bundle level due to different number/type of FUs, or different latencies
- NOPs can be very common (average parallelism in sequential code is limited), so the machine code can be huge
- Solution: code compression (makes the fetch stage more complex)
A VLIW Processor: Lx

- Designed for low-power, high-performance applications
- 4-issue VLIW, max 2 mult, 1 memory and 1 branch instruction per bundle
- See Faraboschi et al., “Lx: a technology platform for customizable VLIW embedded processing”, ACM ISCA‘00
Multiple Cluster VLIW

May be composed of 1 to 4 clusters, with a single program counter.
Instruction Scheduling

- Instruction Scheduling techniques are required in VLIW machines, and to a lesser extent in Superscalars, to exploit ILP and LLP.

- It is necessary to map the instructions over the machine functional units.

- This mapping must account for time constraints and dependencies among the tasks.

- The goal is to minimize the total execution time for the program.
  - It is often necessary to pursue this goal indirectly, since it is difficult to derive optimizing policies from it.
The Scheduling Problem

- Scheduling as a general problem
  - Our Scheduling problem is a specialization of a problem common to many disciplines, including Logistics and many areas of Engineering
  - As a problem of optimization, Scheduling is tackled by Operations Research
  - Techniques from Operations Research are therefore used in the solution of our specialized problem
  - We also inherit complexity results: the Scheduling problem is NP-hard
Scheduling Basic Blocks

- A *Basic Block* is a portion of code with single entry and single exit points.

- Within a basic block, the schedule is constrained only by:
  - Data Dependencies: an instruction cannot be executed if its operands are not ready.
  - Machine Resources: a functional unit must be available to handle an instruction.
Several heuristic algorithms have been proposed to solve the Scheduling problem within Basic Blocks:

- Instruction schedulers (for each cycle, issue as many instructions as possible from a set of ready instructions)
- Operation schedulers (for each instruction, issue it at the most favorable cycle)

Both types of algorithm can schedule instructions with an As Soon As Possible or As Late As Possible policy

Resource reservation tables or automata are used to represent machine resources
ASAP Instruction scheduling

- Create the data dependency graph
- At each step:
  - Detect candidate for scheduling
  - Schedule them based on heuristic criteria, up to capacity of the VLIW bundle
  - Remove the scheduled operations from the graph
  - Add a new, empty bundle to the schedule
ASAP Operation scheduling

- Create the data dependency graph

- At each step:
  - Select candidate instruction based on heuristic criteria
  - Schedule it in the earliest possible cycle
  - Remove it from the graph
Beyond Basic Blocks

- Basic Blocks do not display, on average, a large amount of parallelism.
- A classic study (by Tjaden and Flynn) shows that the degree of parallelism within Basic Blocks is less than 3 for most applications.
- We need to find and exploit other sources of parallelism, such as loops, or to create larger blocks to work on.
Barriers to Scheduling

*Barriers* are imposed to scheduling by the control flow

- Branch Barrier: an instruction cannot be moved beyond a branch or join point
- Loop Barrier: an instruction cannot be moved between two iteration of the same loop

Various techniques exist to bypass these barriers:

- Branch Barrier: Trace Scheduling, Superblock Scheduling, If-Conversion
- Loop Barrier: Loop Unrolling, Modulo Scheduling
Trace Scheduling

- Trace scheduling focuses on *traces*
- A trace is a *loop-free sequence of basic blocks embedded in the control flow graph* (Fisher)
- It is an execution path which can be taken for some set of input
- The chances that a trace is actually executed depend on the input set that allow its execution
- Some traces are executed much more frequently than others
Trace Scheduling

- Trace scheduling schedules traces in order of decreasing probability of being executed
  - So, most frequently executed traces got scheduled better
  - Traces are scheduled as if they were basic blocks, so there is no special consideration for branches
  - This may force the compiler to generate off-schedule instruction to preserve program semantics
Trace Scheduling

- Trace scheduling & Loops
  - Trace scheduling cannot proceed beyond a loop barrier (i.e. a back edge in the control flow graph)
  - Techniques used to overcome this limitation are based on loop unrolling.

- Negative effects of unrolling
  - Unrolling produces much extra code
  - It also loses performance, because of the costs of starting and closing the iterations
Trace Scheduling: Exercise

- Trace Selection
- Primary Trace Scheduling
- Bookkeeping
Trace Selection

Red arrows show the most frequent execution trace
Black arrows show the less frequent execution trace
The data dependency graph constrains the scheduler
## Resource use & Priorities

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency</th>
<th>FU</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>ALU</td>
<td>6</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>Mul</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>ALU</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>LS</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>ALU</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>2</td>
<td>LS</td>
<td>8</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>ALU</td>
<td>3</td>
</tr>
<tr>
<td>H</td>
<td>2</td>
<td>LS</td>
<td>2</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td>ALU</td>
<td>8</td>
</tr>
<tr>
<td>J</td>
<td>1</td>
<td>ALU</td>
<td>7</td>
</tr>
<tr>
<td>K</td>
<td>1</td>
<td>ALU</td>
<td>1</td>
</tr>
<tr>
<td>L</td>
<td>4</td>
<td>Mul</td>
<td>6</td>
</tr>
</tbody>
</table>
The front of instructions free from data dependencies is added to the Ready-Set.

Highest-priority instructions ($I=8$, $F=8$, $A=6$, $D=3$) are scheduled at current cycle (1).

Resource constraints do not allow the issue of $D$ and $A$. 

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Ready-Set</th>
<th>ALU</th>
<th>LS</th>
<th>Mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I F D A</td>
<td>I</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>I</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The front of instructions free from data dependencies is added to the Ready-Set.

Highest-priority instructions ($J=7$, $A=6$, $D=3$) are scheduled at current cycle (2).

Resource constraints do not allow the issue of $D$ and $A$. 

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Ready-Set</th>
<th>ALU</th>
<th>LS</th>
<th>Mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I F D A</td>
<td>I</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>D A J</td>
<td>J</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Trace Scheduling - 3

The front of instructions free from data dependencies is added to the Ready-Set

Highest-priority instructions (A=6, L=6, D=3, G=3, K=1) are scheduled at current cycle (3)

Resource constraints do not allow the issue of G and K
Trace Scheduling - 4

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Ready-Set</th>
<th>ALU</th>
<th>LS</th>
<th>Mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I F D A</td>
<td>I</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>D A J</td>
<td>J</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>D A K L G</td>
<td>A</td>
<td>D</td>
<td>L</td>
</tr>
<tr>
<td>4</td>
<td>K G</td>
<td>G</td>
<td>D</td>
<td>L</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- No new instructions are added to the Ready-Set
- Highest-priority instructions ($G=3$, $K=1$) are scheduled at current cycle (4)
- Resource constraints do not allow the issue of $K$
Instruction E is freed and added to the Ready-Set

Highest-priority instructions ($K=1$, $E=1$) are scheduled at current cycle (5)

Resource constraints do not allow the issue of E (could have issued E instead of K, though)
Trace Scheduling - 6

No new instructions are added to the Ready-Set

Instruction E is scheduled at current cycle (6)
Instruction H is freed and added to the Ready-Set

Instruction H is scheduled at current cycle (7)
<table>
<thead>
<tr>
<th>Cycle</th>
<th>Ready-Set</th>
<th>ALU</th>
<th>LS</th>
<th>Mul</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I F D A</td>
<td>I</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>D A J</td>
<td>J</td>
<td>F</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>D A K L G</td>
<td>A</td>
<td>D</td>
<td>L</td>
</tr>
<tr>
<td>4</td>
<td>K G</td>
<td>G</td>
<td>D</td>
<td>L</td>
</tr>
<tr>
<td>5</td>
<td>K E</td>
<td>K</td>
<td></td>
<td>L</td>
</tr>
<tr>
<td>6</td>
<td>E</td>
<td>E</td>
<td></td>
<td>L</td>
</tr>
<tr>
<td>7</td>
<td>A</td>
<td></td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td>H</td>
<td></td>
</tr>
</tbody>
</table>
Must redo D (moved up the split)
Must change join point to avoid K
Bookkeeping: solution

- Must redo D (moved up the split)
- Must change join point to avoid K
Superblock scheduling

- Variant of Trace Scheduling
- Creates more redundant code
- But removes troublesome join points:
  - Same effect on primary trace
  - Less bookkeeping
- Overall, it is a good solution when more than one frequent trace is present
Superblock formation

- Must redo D (moved up the split)
- Must remove the join point to avoid K
Modulo Scheduling

- An algorithm to schedule instructions beyond the Loop Barrier
- It is based on the Software Pipelining principle
  - Issue an iteration before the previous one has ended its execution
  - A new iteration is issued every $N$ cycles, and the code is rearranged so that instructions in the same position in cycle $i$ and $i+N$ are compatible
  - Actually, all iterations of the same $mod N$ class share this property
Modulo Scheduling - Example

Let us take a simple example:

\[ Z[i] = 2 \times X[i] + Y[i] \]

With the following resources available

<table>
<thead>
<tr>
<th>Load</th>
<th>Store</th>
<th>Add</th>
<th>Mult</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latencies</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Number</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Dependence Graph & Schedule

Cycle | Instructions
0     | load X, load Y
1     | *
2
3
4
5     | +
6     | store Z
7
We cannot initiate two loops at the same time, because we would need more load and add units than we actually have.

Therefore, we try initiating a loop at each clock cycle (Initiation Interval = 1)
Initiation Interval = 1

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Iteration 1</th>
<th>Iteration 2</th>
<th>Iteration 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>load X, load Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>*</td>
<td>load X, load Y</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td></td>
<td>load X, load Y</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>store Z</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>store Z</td>
<td>+</td>
</tr>
</tbody>
</table>
Comments

- This first schedule, with $II = 1$, fails, because at time 7 we would need 2 store units, while we have only one.
- We try another schedule, with an *Initiation Interval* of 2
<table>
<thead>
<tr>
<th>Cycle</th>
<th>Iteration 1</th>
<th>Iteration 2</th>
<th>Iteration 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>load X, load Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>load X, load Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>store Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>store Z</td>
<td></td>
</tr>
</tbody>
</table>

Initiation Interval = 2
Comments

- Now the schedule is feasible, and our loop is successfully scheduled with $II = 2$
- Had we obtained an unfeasible schedule, we would have iterated the scheduling procedure with growing Initiation Intervals
Research Topics

- Compiler Construction
  - Compiler Optimizations
  - Front-End Construction

- Instruction Scheduling
  - Dynamic Scheduling for Java on VLIW systems
  - Static Scheduling for high-ILP clustered architectures

- Compiler Optimization meets Computer Architecture Design
  - Exploration of Design Space: Compiler + Architecture
  - Hardware Design Languages: Synthesis as Compilation
Compiler Construction

- Compiler Optimizations
  - Compiler optimization with SUIF
  - Compiler optimization with GCC 3.x
- Front-End Construction
  - Basic Front-End Construction: C front-end for SUIF
  - New programming language implementation: C# front-end
Instruction Scheduling

- Dynamic Scheduling for Java on VLIW systems
- Dynamic Trace Scheduling/Software Pipelining
- Real-Time Scheduling

- Static Scheduling for high-ILP clustered architectures
  - Graph Clustering as a general Optimization Problem (genetic solution?)
  - Graph Clustering as a specific problem (domain-specific heuristics?)
  - Specific Program Transformation and Optimization techniques
Exploration of Design Space

- Compiler Transformations: Loop Fusion, Loop Tiling, Loop Unrolling
- Architectural Design: cache parameters, parallelism, etc.
- Exploration of the Design Space:
  - Optimization Problem: Genetic algorithms, Simulated Annealing, etc.
  - Power and Performance estimation: analytical models, simulation
  - Exploration Framework: robust tools to link compiler, simulator, and exploration
Hardware Design Languages

- **Java as an HDL**
  1. Polymorphism and dynamic class loading to express reconfigurability
  2. Java to SystemC synthesis

- **UML as an HDL**
  1. Modelling of HW features in UML
  2. UML to SystemC synthesis

- **Metrics for HW/SW codesign**: compiler techniques to extract metrics from SystemC (or VHDL, etc.) designs
Contact Details

Giovanni Agosta
e-mail: agosta@elet.polimi.it
tel: 02/2399-3550
http://www.elet.polimi.it/upload/agosta/