Code Scheduling Techniques for VLIW Processors and Their Applicability to Clustered Architectures

Anup Gangwar

Embedded Systems Group,
Department of Computer Science and Engineering,
Indian Institute of Technology Delhi

http://embedded.cse.iitd.ernet.in

March 11, 2004
Outline

- Processor Architectures: RISC, SuperScalar and VLIW
- Clustered VLIW Processors
- Intermediate Representations for VLIW Compilers
- Block Formation Techniques
- Acyclic Code Scheduling in VLIW Compilers
- Cyclic Code Scheduling in VLIW Compilers
- Acyclic Code Scheduling in Clustered VLIW Compilers
- Cyclic Code Scheduling in Clustered VLIW Compilers
- Conclusions
A Pipelined RISC processor can execute at most one Instruction per cycle (IPC)

Typical hazards such as branches and cache misses reduce IPC to less than one

Advantages:
- Simplified hardware and compiler
- Low power consumption

Disadvantages:
- Low performance

Increase in performance beyond 1 IPC can be achieved by multiple-issue processors

Most of the current embedded processors are RISCs: ARM, MIPS, StrongARM etc.
SuperScalar Processor Architecture

- Have multiple functional units (ALUs, LD/ST, FALUs etc.)
- Multiple instruction executions may be in progress at the same time
- Detect parallelism dynamically at run-time

Advantages:
- Binary compatibility across all generations of processors
- Compilation is trivial, at most compiler can rearrange instructions to facilitate detection of ILP at run-time

Disadvantages:
- High power consumption
- Complicated hardware: hence not very suitable for customization

Most of the General Purpose Processors are SuperScalars: Pentium (Pro, II, III, 4), UltraSPARC, Athlon, MIPS10K etc.
VLIW Processor Architecture

- Compiler extracts parallelism, these have evolved from horizontal microcoded architectures.
- Latest industry coined acronym, **EPIC** for Explicitly Parallel Instruction Computing.

Commercial Architectures:
- General Purpose Computing: Intel Itanium
- Embedded Computing: TriMedia, TiC6x, Sun’s MAJC etc.

<table>
<thead>
<tr>
<th>RISC</th>
<th>SuperScalar</th>
<th>VLIW (4 issue)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD r1, r2, r3</td>
<td>ADD r1, r2, r3</td>
<td>ADD r1, r2, r3</td>
</tr>
<tr>
<td>SUB r4, r2, r3</td>
<td>SUB r4, r2, r3</td>
<td>SUB r4, r2, r3</td>
</tr>
<tr>
<td>MUL r5, r1, r2</td>
<td>MUL r5, r1, r2</td>
<td>MUL r5, r1, r2</td>
</tr>
</tbody>
</table>
VLIW Processor Architecture (contd)

- Advantages:
  - Simplified hardware: Suitable for customization
  - Less power consumption as compared to SuperScalar processors
  - High performance

- Disadvantages:
  - Complicated compiler: limits retargetability
  - Code size blow up due to explicit NOPs.
Outline

- Processor Architectures: RISC, SuperScalar and VLIW
- Clustered VLIW Processors
- Intermediate Representations for VLIW Compilers
- Block Formation Techniques
- Acyclic Code Scheduling in VLIW Compilers
- Cyclic Code Scheduling in VLIW Compilers
- Acyclic Code Scheduling in Clustered VLIW Compilers
- Cyclic Code Scheduling in Clustered VLIW Compilers
- Conclusions
Clustered VLIW Processors

- For N functional units connected to a RF the Area grows as \( N^3 \), Delay as \( N^{3/2} \) and Power as \( N^3 \) (Rixner et. al, HPCA 2000)
- Solutions is to break up the RF into a set of smaller RFs
Clustered VLIW Architectures

RF-to-RF

[Diagram of clustered VLIW architectures with ALU and LD/ST units]
Clustered VLIW Architectures

Write Across (2)
Clustered VLIW Architectures

Read Across (2)

Ph.D. Thursday Seminar Series
http://embedded.cse.iitd.ernet.in
Slide 9/51
Clustered VLIW Architectures

Write/Read Across (2)
Compilation for Clustered VLIWs

- Compilation is complicated due to partial connectivity between clusters
- Important acyclic and cyclic (modulo) scheduling techniques developed for monolithic VLIWs are not directly applicable
- **Operation ⇒ FU in Cluster** binding problem is the most critical
- Another problem is register allocation
- Scheduling/Binding techniques developed are inter-cluster interconnect specific
Outline

- Processor Architectures: RISC, SuperScalar and VLIW
- Clustered VLIW Processors
- Intermediate Representations for VLIW Compilers
- Block Formation Techniques
- Acyclic Code Scheduling in VLIW Compilers
- Cyclic Code Scheduling in VLIW Compilers
- Acyclic Code Scheduling in Clustered VLIW Compilers
- Cyclic Code Scheduling in Clustered VLIW Compilers
- Conclusions
Block Diagram of a Typical VLIW Compiler

Compiler Front-end

C-Source

* Input parsing

High-Level IR

* High-level code transformations
* Function inlining
* Profiling
* IR conversion

Compiler Back-end

Assembly

* Acyclic scheduling
* Modulo scheduling
* Register allocation
* Post-pass scheduling

Low-Level IR

* Block formation
* ILP enhancement
* Classical optimizations
High Level Intermediate Representation

- **SUIF2**
  - Designed to capture high-level information
  - No instructions, only statements and expressions
  - Statements: if-else, do-while etc.
  - Expressions: Binary expressions etc.

- **Hcode (IMPACT)**
  - Designed to capture high-level information
  - Similar to SUIF2, contains statements mixed with expressions
  - No notion of instructions, enables control-flow profiling
Low Level Intermediate Representation

- MachSUIF
  - Designed to capture low-level information, which is architecture specific
  - Interfaces for target specific code generation, register allocation, scheduling

- Lcode (IMPACT)
  - Designed to capture low-level information, for a typical VLIW architecture
  - Lcode nodes (instructions), support predication etc.
  - Enables block formation, register allocation, code scheduling etc.
Outline

- Processor Architectures: RISC, SuperScalar and VLIW
- Clustered VLIW Processors
- Intermediate Representations for VLIW Compilers
  - Block Formation Techniques
  - Acyclic Code Scheduling in VLIW Compilers
  - Cyclic Code Scheduling in VLIW Compilers
  - Acyclic Code Scheduling in Clustered VLIW Compilers
  - Cyclic Code Scheduling in Clustered VLIW Compilers
- Conclusions
Code Snippet:

```c++
if( i == 10 ){
    j = 100;
    k = 110;
} else{
    j = 200;
    k = 210;
}
```

<table>
<thead>
<tr>
<th>Basic Blocks:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BB-1</strong></td>
</tr>
<tr>
<td>j = 100;</td>
</tr>
<tr>
<td>k = 110;</td>
</tr>
</tbody>
</table>

- Single entry single exit blocks
- No control flow information, representible as DFGs
- Typically very small, large in few media applications
Code Snippet:

```plaintext
....
    a = 50;
    if( i = 10 ){
        j = 100;
        k = 110;
    }else{
        j = 200;
        k = 210;
    }
}
c = a + j + k;
```

Trace:
(when **then** part is more frequently executed):

```
<table>
<thead>
<tr>
<th>Trace-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 50;</td>
</tr>
<tr>
<td>j = 100;</td>
</tr>
<tr>
<td>k = 110;</td>
</tr>
<tr>
<td>c = a + j + k;</td>
</tr>
</tbody>
</table>
```

- Single entry single exit blocks, list scheduling after trace detection
- Bookkeeping code added to preserve semantics
- No speculative execution (in original Fisher approach)
Block Formation Techniques: HyperBlock

- Require support from hardware in the form of predicated execution
- Assume a predicate (TRUE/FALSE) value is appended to each instruction as argument e.g. ADD r1, r2, r3 → ADD r1, r2, r3, p1
- Each instruction execution is conditional on either the predicate being TRUE or FALSE
HyperBlocks (contd)

Code Snippet:

```c
....
a = 50;
if( i = 10 ){
j = 100;
k = 110;
}else{
j = 200;
k = 210;
}
c = a + j + k;
```

HyperBlock:

<table>
<thead>
<tr>
<th>HyperBlock-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 50, TRUE</td>
</tr>
<tr>
<td>p&lt;t&gt; = ? i = 10;</td>
</tr>
<tr>
<td>j = 100, p&lt;t&gt;</td>
</tr>
<tr>
<td>k = 110, p&lt;t&gt;</td>
</tr>
<tr>
<td>j = 200, p&lt;f&gt;</td>
</tr>
<tr>
<td>k = 210, p&lt;f&gt;</td>
</tr>
<tr>
<td>c = a + j + k, TRUE</td>
</tr>
</tbody>
</table>

- Single entry multiple exit blocks, incorporates both IF and ELSE parts
- Specialized scheduling techniques after detection, tend to grow large
- Performance may not increase due to dynamic nullification
Block Formation Techniques: SuperBlock

- Superblocks are traces with all join points removed via tail duplication
- May lead to a larger code than trace
- Enable easy application of certain optimizations such as copy propagation
- Require hardware support for speculative execution
SuperBlocks (contd)

Code Snippet:

```java
...
    a = 50;
    if( i = 10 ){
        j = 100;
        k = 110;
    }else{
        j = 200;
        k = 210;
    }
    c = a + j + k;
```

SuperBlocks:
(when **then** part is more frequently executed):

<table>
<thead>
<tr>
<th>SuperBlock-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = 50;</td>
</tr>
<tr>
<td>j = 100;</td>
</tr>
<tr>
<td>k = 110;</td>
</tr>
<tr>
<td>c = a + j + k;</td>
</tr>
</tbody>
</table>

- Single entry multiple exit blocks
- Specialized scheduling after detection
Block Formation Techniques: Treegion

Code Snippet:

```java
....
if ( i = 10 ){
    j = 100;
    k = 110;
}else{
    j = 200;
    k = 210;
}
c = j + k;
```

Treeregions:

<table>
<thead>
<tr>
<th>Treegion-1</th>
</tr>
</thead>
</table>
| if ( i = 10 ){
    j = 100;
    k = 110;
}else{
    j = 200;
    k = 210;
} |

- Single entry multiple exit block (a tree of BBs)
- Global scheduling is possible, support for speculative execution
- Profile information is not needed
# Comparison of Block Formation Techniques

<table>
<thead>
<tr>
<th>Name</th>
<th>If-Else Included</th>
<th>Entry</th>
<th>Exits</th>
<th>H/W Supp.?</th>
<th>Profile Info.?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic Block</td>
<td>N.A.</td>
<td>Single</td>
<td>Single</td>
<td>None</td>
<td>No</td>
</tr>
<tr>
<td>Traces</td>
<td>Only One</td>
<td>Single</td>
<td>Single</td>
<td>None</td>
<td>Yes</td>
</tr>
<tr>
<td>HyperBlock</td>
<td>Both</td>
<td>Single</td>
<td>Multiple</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>SuperBlock</td>
<td>One or Both</td>
<td>Single</td>
<td>Multiple</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Treegion</td>
<td>Global</td>
<td>Single</td>
<td>Multiple</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
Outline

- Processor Architectures: RISC, SuperScalar and VLIW
- Clustered VLIW Processors
- Intermediate Representations for VLIW Compilers
- Block Formation Techniques
- Acyclic Code Scheduling in VLIW Compilers
- Cyclic Code Scheduling in VLIW Compilers
- Acyclic Code Scheduling in Clustered VLIW Compilers
- Cyclic Code Scheduling in Clustered VLIW Compilers
- Conclusions
Acyclic Code Scheduling: List Scheduling

- Works on a Data Flow Graph (DFG)
  1. Sort noted based on some priority function
  2. Schedule as many nodes in this cycle as possible
  3. Increase the schedule step and repeat 2, 3 till all nodes are scheduled

- Works well for DFGs, modifications are needed to work with HyperBlocks and SuperBlocks.
Acyclic Code Scheduling: Trace Scheduling

- Works on traces
  1. Do a trace formation on the region of interest
  2. Perform list scheduling on the trace
  3. Add compensation code

- It has been superseded by SuperBlock scheduling algorithms
Outline

- Processor Architectures: RISC, SuperScalar and VLIW
- Clustered VLIW Processors
- Intermediate Representations for VLIW Compilers
- Block Formation Techniques
- Acyclic Code Scheduling in VLIW Compilers
- Cyclic Code Scheduling in VLIW Compilers
- Acyclic Code Scheduling in Clustered VLIW Compilers
- Cyclic Code Scheduling in Clustered VLIW Compilers
- Conclusions
Cyclic Code Scheduling: S/W Pipelining

- Proposed by B. R. Rau in 1981, subsequently modified by Lam (PLDI, 1998), Rau (MICRO, 1994) and others
- Lam’s approach employs hierarchical reduction which is not really relevant today due to HyperBlocks etc.
- Iterative Modulo Scheduling by Rau (MICRO, 1996) is most established

Code Snippet:

```c
for ( i = 0; i < 100; i++ )
    a = a + 1;
```

Architecture has 4 ALUs:

<table>
<thead>
<tr>
<th>Steady State Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD.0 a1 = a0 + 1;</td>
</tr>
<tr>
<td>ADD.1 a3 = a2 + 1;</td>
</tr>
<tr>
<td>ADD.2 a5 = a4 + 1;</td>
</tr>
<tr>
<td>ADD.3 a7 = a6 + 1;</td>
</tr>
</tbody>
</table>
Outline

- Processor Architectures: RISC, SuperScalar and VLIW
- Clustered VLIW Processors
- Intermediate Representations for VLIW Compilers
- Block Formation Techniques
- Acyclic Code Scheduling in VLIW Compilers
- Cyclic Code Scheduling in VLIW Compilers
- Acyclic Code Scheduling in Clustered VLIW Compilers
- Cyclic Code Scheduling in Clustered VLIW Compilers
- Conclusions
Acyclic Code Scheduling: Problems

- Cluster Assignment
- Register Allocation
- Instruction Scheduling
Acyclic Code Scheduling: Jacome et. al.

- Works on DFGs for RF-to-RF architectures
- Divide the DFG into a collection of *Vertical and Horizontal Aggregates*
- A derived value, *Load*, is used to determine load if a particular operation is scheduled onto a particular cluster
- The remaining problem is to map these aggregates onto the clustered architecture, an aggregate is scheduled onto a cluster as a whole and is not further subdivided
- Algorithm can be used for both design-space-exploration and code generation
Vertical and Horizontal Aggregates:

![Diagram showing vertical and horizontal aggregates with nodes labeled 1 to 8 and areas labeled VA-1, VA-2, HA-1, HA-2.]
Acyclic Code Scheduling: Sanchez et. al.

- Works for DFGs for RF-to-RF architectures
- Greedy algorithm which tries to minimize communication amongst clusters
- Simultaneous cluster assignment and scheduling for each operation
- Register assignment is trivial, with generated values first going to the cluster in which this operation is scheduled
Acyclic Code Scheduling: PCC

- Partial Component Clustering, proposed by G. Desoli, 1998
- Works for DFGs and RF-to-RF architectures
- Problem reduction is achieved by identifying Sub-DAGs which are in turn scheduled and bound
- Works well if the Sub-DAGs are balanced in terms of number of operations and schedule lengths
PCC (contd)

DAG of Matrix Init After Loop Unrolling:
Acyclic Code Scheduling: RAW

The RAW Architecture:

Each processor contains ALU, Instruction/Data memory and a switch, which has its own instruction memory.

The RAW Microprocessor
Unlike a VLIW processor, the RAW architecture has multiple flows of control.

- The instruction partitioner, partitions the instruction into multiple groups, both at local and global levels.
- The clustering and merging phases bring down the number of such groups to the number of processing resources.
- The RAW scheduler, schedules these groups onto each processor and generates the required communication operations.
Acyclic Code Scheduling: CARS

- Single step cluster assignment, register allocation and instruction scheduling avoids back-tracking and leads to better overall schedules
- Works for any region (Basic-Blocks, HyperBlocks, SuperBlocks, Treegeons) and RF-to-RF architectures
- Register allocation is performed on the fly to incorporate effects of generated spill code
Acyclic Code Scheduling: TTS

An Example Treeregion:
The TTS Algorithm:

1. Sort basic blocks as per the execution frequency
2. Start list scheduling from the root
3. Consider speculative execution similar to SuperBlocks
4. Repeat till all basic blocks have been scheduled
Works on DFGs for Write Across type of architectures (TiC6201)

Operation partitioning is based on simple simulated annealing approach with cost as schedule length of this DFG accounting for the copy operations

Finally a list scheduling is carried out on the DFG

100 node DFG partitioned in 10 CPU seconds on a SUN Ultra-1, though this could be prohibitively large for big DFGs
**Acyclic Code Scheduling: Our Approach**

- DFGs are build out of the execution traces
- Operation partitioning amongst clusters is carried out by assuming a best neighbour connectivity amongst clusters
- Finally a resource constrained list scheduling is carried out to generate correct schedule
- Very time consuming and not suitable for code generation, only for evaluation of architectures
## Comparison of Acyclic Scheduling Algorithms

<table>
<thead>
<tr>
<th>Name</th>
<th>Region</th>
<th>Architecture(s)</th>
<th>Run-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacome et. al.</td>
<td>DFG, DFG</td>
<td>RF-to-RF</td>
<td>May be high</td>
</tr>
<tr>
<td>Sanchez et. al.</td>
<td>DFG</td>
<td>RF-to-RF</td>
<td>Fast</td>
</tr>
<tr>
<td>PCC</td>
<td>DFG</td>
<td>RF-to-RF</td>
<td>May be high</td>
</tr>
<tr>
<td>RAW</td>
<td>Global</td>
<td>MIT RAW</td>
<td>May be high</td>
</tr>
<tr>
<td>CARS</td>
<td>All</td>
<td>RF-to-RF</td>
<td>Low</td>
</tr>
<tr>
<td>TTS</td>
<td>Treecgion</td>
<td>RF-to-RF</td>
<td>Low</td>
</tr>
<tr>
<td>Leupers</td>
<td>DFG</td>
<td>Write Across</td>
<td>May be high</td>
</tr>
<tr>
<td>Ours</td>
<td>DFG</td>
<td>All</td>
<td>High</td>
</tr>
</tbody>
</table>
Outline

- Processor Architectures: RISC, SuperScalar and VLIW
- Clustered VLIW Processors
- Intermediate Representations for VLIW Compilers
- Block Formation Techniques
- Acyclic Code Scheduling in VLIW Compilers
- Cyclic Code Scheduling in VLIW Compilers
- Acyclic Code Scheduling in Clustered VLIW Compilers
- Cyclic Code Scheduling in Clustered VLIW Compilers
- Conclusions
Cyclic Code Scheduling: CALIBER

Works on DFGs for RF-to-RF architectures, proposed by Jacome et. al, 2002

Tries to achieve final scheduling latencies which are close to monolithic VLIWs

Kernel detection tries to minimize the schedule length as well as register pressure and code size

The retiming module is an extension over the previous modulo scheduling algorithms

The graph partitioning module generates a number of solutions based on their acyclic scheduling algorithm which are heuristically pruned
Cyclic Code Scheduling: Sanchez et. al.

- Works on DFGs for RF-to-RF architectures
- Kernel detection, spill code generation, register allocation and cluster assignment are performed simultaneously
- Similar to monolithic VLIW modulo scheduling algorithms apart from the integration of above modules
Another algorithm from Sanchez et. al., MICRO, 2000 takes care of address generation for multiple-banked memory architectures

Few others but similar to CALIBER and Sanchez et. al. approaches
### Comparison of Cyclic Code Scheduling Algorithms

<table>
<thead>
<tr>
<th>Name</th>
<th>Region</th>
<th>Architecture(s)</th>
<th>Run-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacome et. al.</td>
<td>DFG</td>
<td>RF-to-RF</td>
<td>Medium</td>
</tr>
<tr>
<td>Sanchez et. al.</td>
<td>DFG</td>
<td>RF-to-RF</td>
<td>Medium</td>
</tr>
<tr>
<td>Others</td>
<td>DFG</td>
<td>RF-to-RF</td>
<td>-</td>
</tr>
</tbody>
</table>
Outline

- Processor Architectures: RISC, SuperScalar and VLIW
- Clustered VLIW Processors
- Intermediate Representations for VLIW Compilers
- Block Formation Techniques
- Acyclic Code Scheduling in VLIW Compilers
- Cyclic Code Scheduling in VLIW Compilers
- Acyclic Code Scheduling in Clustered VLIW Compilers
- Cyclic Code Scheduling in Clustered VLIW Compilers
- Conclusions
Conclusions: Where does all this take us?

- All scheduling techniques are for RF-to-RF architectures except one (Rainer Leupers, PACT 2000)
- Interconnects influence the scheduling algorithms to a large extent, RF-to-RF is the simplest to deal with, but has the lowest performance (Gangwar et. al., WASP, 2003)
- Solutions for clustered architectures for modulo scheduling are trivial and a comparative study is missing along with concrete running times
- Future work will focus on developing scheduling algorithms for architectures with specialized interconnects
Thank You