

VU2 185.324

Compilation Techniques for VLIW Architectures

Dietmar Ebner ebner@complang.tuwien.ac.at
Florian Brandner brandner@complang.tuwien.ac.at

<http://complang.tuwien.ac.at/cd/vliw>

Last Lectures (1)

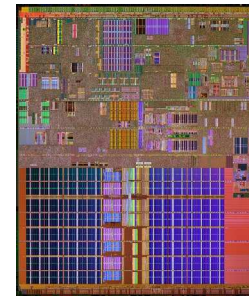
- Instruction Scheduling
 - List Scheduling
 - forward
 - backward
 - Alternative Approaches
 - Resource Models
 - Reservation tables
 - Finite state automata

Last Lectures (2)

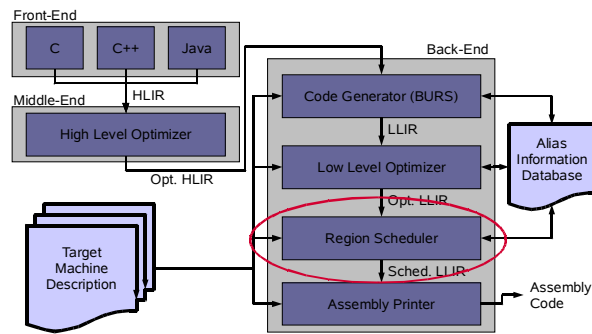
- Scheduling for VLIW architectures
 - Trace Scheduling
 - Trace Selection
 - Handling Loops
 - Code Motion / Compensation Code

In Today's Lecture

- Different Types of Regions
 - Superblocks
 - Hyperblocks
 - Treeregions
- Region Enlargement Techniques



Phases of an ILP Oriented Compiler



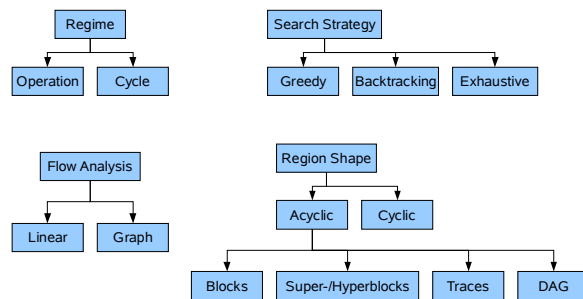
05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #5

Instruction Scheduling

- Most fundamental ILP-oriented phase
- VLIW Scheduling
 - Identify and group operations that can be executed in parallel
 - Minimize schedule length
 - Obey data dependencies and resource limitations
 - Cyclic vs. acyclic scheduling
 - Phase ordering issues

05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #6

Classification



05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #7

Scheduling Phases

- “Global” scheduling is too complex in general
- Basic blocks do not offer sufficient amounts of ILP for wide-issue machines
- VLIW architectures often include hardware support
- Phase Ordering
 - Region formation
 - Schedule compaction

05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #8

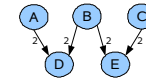
Schedule Compaction

- List scheduling widely used in practice
- Operates on the data dependence graph
 1. Select and schedule a *ready* node from the DDG
 2. Repeat until the DDG is empty
- A node is *available* if all predecessors in the DDG have already been scheduled
- A node is *ready* if it is available, and all hardware constraints are met

05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #9

List Scheduling - Limitations

- How to select from the list of ready nodes?

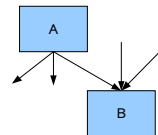


- List Schedulers often tend to be too *greedy*
 - Operations that occupy a resource for multiple cycles might be scheduled too early, preventing subsequent critical instructions to become ready
 - Operations scheduled too early might unnecessarily increase register pressure

05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #10

Region Selection

- Iterative *trace growing* using the **mutual most likely** strategy



- Stops, whenever no mutual most likely blocks can be found or a backedge is encountered

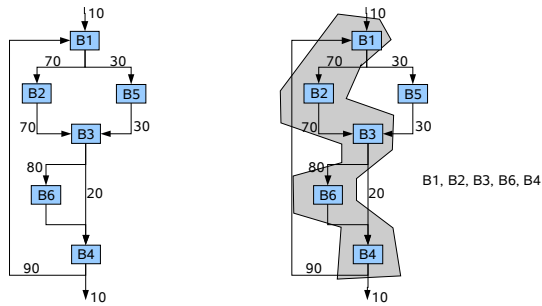
05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #11

Traces

- Definition
 - Linear sequence of blocks with possibly multiple entrances and exits
 B_0, B_1, \dots, B_n
 - Formal Properties
 - (1) Each basic block is a predecessor of the next on the list
 - (2) For any i and k , there is no path $B_i \rightarrow B_k \rightarrow B_i$, except for those passing through B_0
 - Does not prohibit forward branches or control flow leaving and re-entering the region!

05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #12

Example Trace



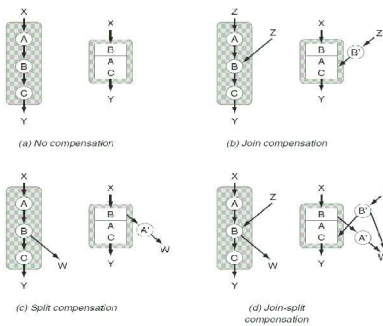
05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #13

Compensation Code

- What happens if we move instructions across join/split points
- Preserve all paths from the original code sequence in the transformed control flow after scheduling
- Compensation code drastically complicates trace based schedulers

05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #14

Compensation Scenarios



05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #15

Superblocks

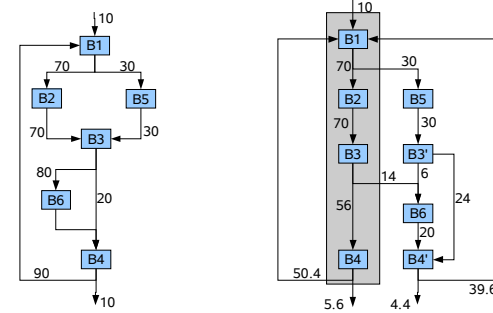
- Superblocks are traces without **side entrances**
- Single-Entry Multiple-Exit (SEME)
- Formal Properties
 - (1) Each basic block is a predecessor of the next on the list
 - (2) For any i and k , there is no path $B_i \rightarrow B_k \rightarrow B_i$, except for those passing through B_0
 - (3) There may be no branches into a block in the region, except to B_0

05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #16

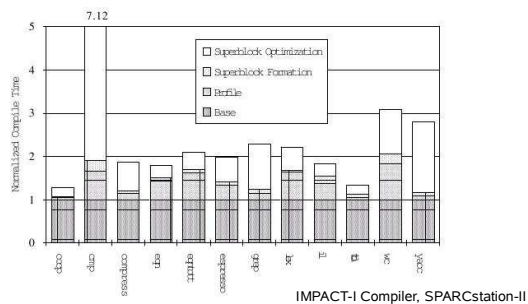
Tail Duplication

- Additional restrictions facilitate superblock schedulers
- Stopping trace formation at every join point is – at the first glance – a severe restriction
- Tail duplication can be used to create a copy of the rest of the trace
- Tail duplication can be seen as a form of “compensation code” that is created before the schedule compaction phase

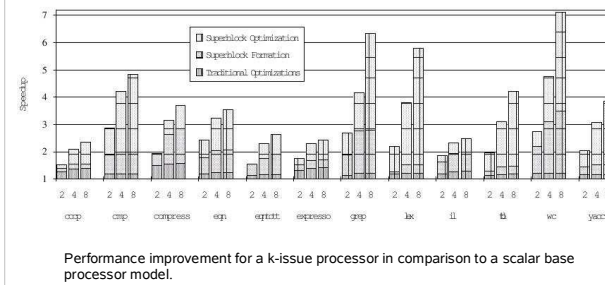
Superblock Formation



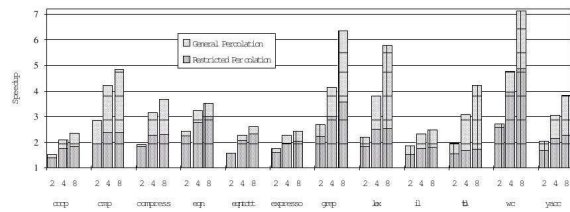
Experimental Results



Performance Improvement



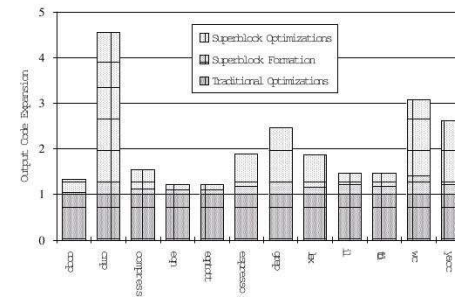
Effect of Speculation



Restricted Percolation: no support for disregarding exceptions for load, store, divide, and floating point instructions. Non-trapping versions of those instructions are included in the "General Percolation" model.

05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #21

Code Size Expansion



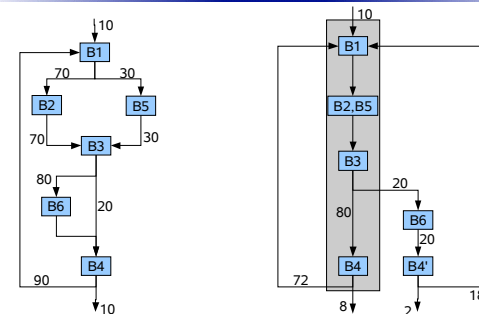
05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #22

Hyperblock

- SEME regions with internal control flow
- Relaxed variant of superblocks
- Employ predication to fold multiple control paths into a single superblock
 - Allows to create traces with higher execution probability
 - Removes side exits and its schedule constraints

05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #23

Hyperblock Example

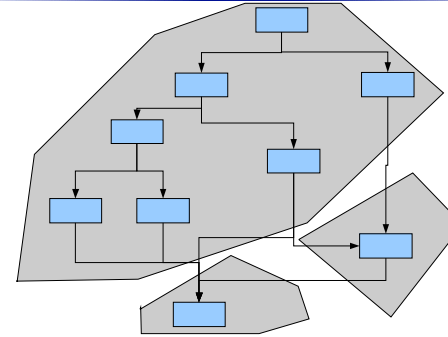


05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #24

Treeregions

- Tree of basic blocks within the CFG
- List of blocks B_0, B_1, \dots, B_n such that each B_j except for B_0 has exactly one predecessor B_i with $i < j$
- Any path through the treeregion yields a superblock
- Tail duplication is used to enlarge treeregions just like in the superblock case
- Often referred as “*non-linear regions*” (c.f. “*linear regions*” for traces, superblocks)

Treeregion Example



Region Comparison

	Trace	Superblock	Hyperblock	Treeregion
Year Proposed	1979	1988	1992	1997
Policy at splits	one way, most likely	one way, most likely	predicate when possible	both ways
Policy at joins	continue	stop	stop	stop
Policy at backedges	unrolled loops regarded as essential feature	stop, but apply region enlargement techniques	stop, but apply region enlargement techniques	stop, but apply region enlargement techniques
Proposed measures to increase region size	loop unrolling	tail duplication, peeling, unrolling, and target expansion	predication for rejoin, tail duplication for unpredicated splits, peeling, unrolling, and target expansion	tail duplication, peeling, unrolling, and target expansion

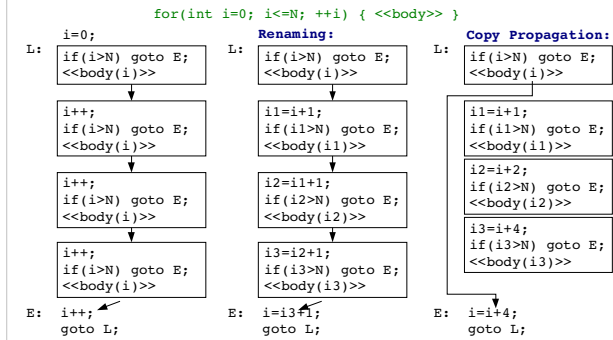
Region Enlargement

- Can be used to trade code size for performance
 - Make extra copies of highly iterated code
- Most common techniques
 - Loop unrolling
 - Loop peeling
 - Branch target expansion
 - (reverse) if-conversion (hyperblocks)

Loop Unrolling

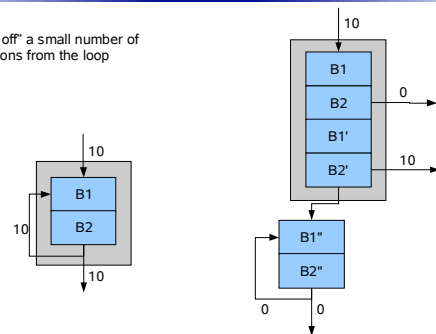
- Duplicate a loop body several times
 - preconditioning / postconditioning to handle trip counts mod n
 - preconditioning is not possible for data dependent loop exits
- Small loops are often completely unrolled
- Can be performed both before and after region formation

Loop Unrolling: Increasing Parallelism

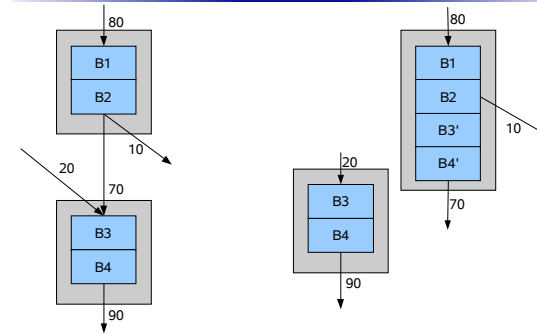


Loop Peeling

"Peel off" a small number of iterations from the loop



Target Expansion



Dependence Removing Optimizations


- Eliminate data dependencies among instructions within frequently executed regions
 - Register Renaming
 - Operation Migration
 - Induction Variable Expansion
 - Accumulator Variable Expansion
 - Operation Combining

05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #33

Register Renaming

- Eliminates anti- and output-dependencies by assigning unique registers to different definitions of the same register.
- Important within individual bodies of an unrolled loop

```
R1 = load[a]
R1 = R1 + c
store a, R1
R1 = load[a+1]
R1 = R1 + c
store a+1, R1
```



```
R1 = load[a]
R1 = R1 + c
store a, R1
R2 = load[a+1]
R2 = R2 + c
store a+1, R2
```

05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #34

Operation Migration

- Moves an instruction from a region where its result is not used to less frequently used regions
- A copy has to be placed at the target region of each exit where the defined variable is live
- All of the data dependencies associated with that instruction can be eliminated from the region

05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #35

Induction Variable Expansion

- Eliminates redefinitions of induction variables within an unrolled loop
- Each definition of the induction variable is given a new register
- Patch code has to be inserted if the induction variable is used outside the region to recover the proper value

05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #36

Accumulator Variable Expansion

- Accumulation operations often define the critical path within loops
- Very similar to induction variable expansion
- Replaces each definition of an accumulator variable with a new register
- Requires additional initialization in the preheader and code that accumulates the partial results at region exits
- Often unsafe for floating point operations

05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #37

Operation Combining

- Eliminates flow dependencies among pairs of instructions with the same precedence and with a compile-time source operand
- Often arise between address calculations and memory access instructions

$p = a + 1$
 $R1 = \text{load}[p]$
 $q = p + 1$
 $R2 = \text{load}[q]$



$p = a + 1$ $q = a + 2$
 $R1 = \text{load}[p]$ $R2 = \text{load}[q]$

05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #38

Outlook

- Cyclic Scheduling
 - Software Pipelining
 - Modulo Scheduling
- Data- / Control Speculation
- Predicated Execution

05/27/08 Ebner, Brandner | Compilation Techniques for VLIWs | SS08 Slide #39