
*The IBM Research VLIW
Project*

Kemal Ebcioglu
IBM T.J. Watson Research Center
kemal@watson.ibm.com

Overview of IBM Research VLIW Project

- ◆ Research on parallelization of sequential natured code since 1986
 - ◆ Hardware prototype operational
 - 8 ALU operations
 - 4 load/stores
 - 7 conditional branches
 - ◆ Object code translation project
 - Self-modifying code
 - Precise exceptions
 - Re-ordering memory references with multiprocessor consistency
 - Memory mapped I/O
-

Overview of IBM Research VLIW Project

- ◆ 3rd generation VLIW parallelizer
 - 5X pathlength reduction over POWERPC on integer code, without profiling
 - ◆ Application of VLIW techniques to superscalar compilers
 - Helped obtain 18% improvement in SPECint92 on POWER, POWER-2 and POWERPC 601
-

VLIW vs. Out-of-Order Superscalar

Superscalar

- ◆ Small window for finding parallelism
- ◆ Complex design limits issue rates
- ◆ Executes single path
- ◆ Poor branch throughput
- ◆ Long pipeline, branch stalls

(IBM) VLIW

- ◆ Large window for finding parallelism
 - ◆ Simple design
 - ◆ Executes multiple paths. Resilient to mispredictions
 - ◆ High branch throughput
 - ◆ Short pipeline, zero branch stalls
-

VLIW vs. Out of Order Superscalar

Superscalar

- ◆ Performance difficult to predict (Compiler scheduling can degrade performance)

(IBM) VLIW

- ◆ Predictable performance (Compiler scheduling improves performance)
-

VLIW Evolution

- ◆ Multiflow VLIW machine (ca. 1984)
 - For technical applications, not general purpose
 - High data memory latency, bad for integer code
 - Technology limitations
 - ◆ Did not fit on a chip
 - ◆ Had to use partitioned register files
 - ◆ Memory banks controlled by compiler
 - Old trace scheduling did not do well w/branches
 - Not scalable
 - New, incompatible architecture
-

VLIW Evolution

- ◆ VLIW enablers ca. 1998
 - VLSI density improved significantly
 - New parallelizing compiler techniques discovered for branch intensive integer code
 - ◆ VLIW techniques became applicable to general code
 - Solutions for object code compatibility problem found
 - Approaches to scalable VLIW design discovered
-

Some IBM VLIW Project Contributions

- ◆ New scheduling for branch intensive code, that handles multiple paths directly (unlike trace scheduling + predication)
 - ◆ New software pipelining with variable iteration issue rates
 - ◆ New architectural features for supporting multiple branches/cycle
 - ◆ New architecture/compiler features for speculative execution, out of order memory references, MP strong consistency, memory mapped I/O, object code compatible VLIW, scalable VLIW...
-

A compilation example

- ◆ **loop:**
 - x=f(x)**
 - cc1=test1(x)**
 - if cc1 goto T1**
 - x=g(x,y)**
- T1: cc2=test2(x)**
 - if cc2 goto loop**
- exit: x live**
- ◆ **No parallelism if control dependences are respected**
- ◆ **if test1(x) is true, longest dependence cycle has length 1: x=f(x) (1 cycle/iteration)**
- ◆ **if test1(x) is false, longest dependence cycle has length 2: x=f(x), x=g(x,y) (2 cycles/iteration)**
- ◆ **Our approach generates a variable iteration issue rate: 1 or 2 cycles/iteration**

Object Code Compatibility

Approaches for VLIW

- ◆ Static translation of executables (Silberman-Ebcioglu 92, Sites et al. 93)+use an old engine as backup
 - Put information in new executables to facilitate OCT
 - ◆ Incremental object code translation by HW at icache miss time (Ebcioglu-Groves 90, Franklin-Smoth. 94)
 - ◆ Incremental object code translation by software+hardware assist (similar to T. Conte 95)
 - Requires little special HW for compatibility, plain VLIW
 - Virtual Machine Monitor rapidly translates code on first execution, saves translation.
 - No changes to old architecture, full VLIW benefits
-

POWERPC/VLIW Path Length Ratios

Program	POWERPC Instr/VLIW
compress	5.4
eqntott	7.7
fgrep	7.5
lex	6.0
prof	4.7
sc	4.7
sed	5.8
uncompress	4.2
xlisp	3.9
yacc	4.4
Geometric mean	5.3

No profiling

No inlining

Base opt. level

16 Alus/8W br.

~2.7 billion

VLIW instr.

simulated

Some Publications Related to IBM VLIW

- K. Ebcioglu, R. Groves, K.C. Kim, G. Silberman, I. Ziv. “VLIW Compilation Techniques in a Superscalar Environment” Proc. PLDI 94.
 - T. Nakatani and K. Ebcioglu. “Making Compaction Based Parallelization Affordable” IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 9, pp. 1014-1029, September 1993.
 - S.M. Moon and K. Ebcioglu “A Study on the Number of Memory Ports in Multiple Instruction Issue Machines” Proceedings of MICRO-26, IEEE Press, 1993.
 - S.M. Moon and K. Ebcioglu “On Performance and Efficiency of VLIW and Superscalar” Proceedings of the 1993 International Conference on Parallel Processing, Volume 2, pp. 283-287, CRC Press, Ann Arbor.
 - G. Silberman and K. Ebcioglu, “An Architectural Framework for Supporting Heterogeneous Instruction-Set Architectures,” IEEE Computer, Vol. 26, No. 6, June 1993, pp. 39-56.
 - S.M. Moon and K. Ebcioglu, “An Efficient Resource-Constrained Global Scheduling Technique for Superscalar and VLIW Processors,” Proc. MICRO-25, pp. 55-71, IEEE Press, December 1992.
 - G. Silberman and K. Ebcioglu, “An Architectural Framework for Migration from CISC to Higher Performance Platforms,” Proc. 1992 International Conference on Supercomputing, pp. 198-215, ACM Press, 1992.
 - T. Nakatani and K. Ebcioglu, “Using a Lookahead Window in a Compaction based Parallelizing Compiler,” Proceedings, 23rd Workshop on Microprogramming and Microarchitecture, IEEE and ACM, pp. 57-68, IEEE Computer Society Press, 1990. (This paper won the Workshop’s best paper award).
-

Some Publications Related to IBM VLIW

- K. Ebcioglu and R. Groves, "Some Global Compiler Optimizations and Architectural Features for Improving Performance of Superscalars," Research Report no. RC16145, IBM T.J. Watson Research Center, Yorktown Heights, NY, 1990. (Presented at the ICCD-1990 conference).
 - K. Ebcioglu and T. Nakatani, "A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture," in Languages and Compilers for Parallel Computing, D. Gelernter, A. Nicolau, and D. Padua (eds.), Research Monographs in Parallel and Distributed Computing, pp. 213-229, MIT Press, 1990.
 - U. Schwiegelshohn, F. Gasperoni, and K. Ebcioglu, "On Optimal Parallelization of Arbitrary Loops," in Journal of Parallel and Distributed Computing, 11, 130-134. Academic Press, 1991.
 - K. Ebcioglu and M. Kumar, "A Wide Instruction Word Architecture for Parallel Execution of Logic Programs Coded in BSL," in New Generation Computing, 7 (1990) 219-242, Tokyo, OMSHA Ltd and Springer-Verlag, 1990.
 - T. Nakatani and K. Ebcioglu, "Combining as a Compilation Technique for VLIW Architectures," in Proceedings 22nd Workshop on Microprogramming and Microarchitecture, Dublin, ACM Press, pp. 43-57, 1989. (This paper won the Workshop's co-best paper award).
 - K. Ebcioglu and A. Nicolau, "A Global Resource-constrained Parallelization Technique," in Proceedings Third International Conference on Supercomputing, pp. 154-163, Crete, June 1989.
 - K. Ebcioglu, "Some Design Ideas for a VLIW Architecture for Sequential Natured Software," in Parallel Processing (Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing), pp. 3-21, M. Cosnard et al. (eds.), North Holland, 1988.
-

Additional Foils

IBM VLIW Compilation Techniques

- ◆ Enhanced pipeline scheduling
(technique for software pipelining loops
with conditional jumps)
 - ◆ Moon-Ebcioglu 1992
 - ◆ Ebcioglu-Nakatani 1989
-

Scheduling Cycle 1 & Start of Cycle 2

◆ loop:

```
-----  
x=f(x)  
-----  
cc1=test1(x)
```

```
if cc1 goto T1  
x=g(x,y)
```

```
T1: cc2=test2(x)
```

```
if cc2 goto loop  
exit: x live
```

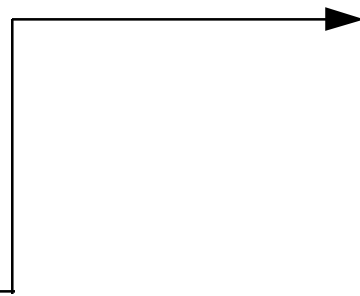
◆ loop:

```
x=f(x)  
-----  
cc1=test1(x)  
x''=g(x,y)
```

```
-----  
if cc1 goto T1  
x=x''
```

```
T1: cc2=test2(x)
```

```
if cc2 goto loop  
exit: x live
```



Cycle 2 op#3

◆ loop:

x=f(x)

cc1=test1(x)

x''=g(x,y)

if cc1 goto T1

x=x''

T1: cc2=test2(x)

if cc2 goto loop

exit: x live

◆ loop:

x=f(x)

cc1=test1(x)

x''=g(x,y)

cc2=test2(x)

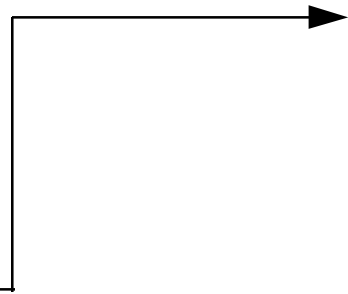
if cc1 goto T2

x=x''

cc2=test2(x)

T2: if cc2 goto loop

exit: x live



Cycle 2 op#4

◆ loop:

```
x=f(x)
-----
cc1=test1(x)
x''=g(x,y)
cc2=test2(x)
```

```
-----
if cc1 goto T2
x=x''
cc2=test2(x)
```

```
T2: if cc2 goto loop
exit: x live
```

◆ loop:

x'=f(x)

L1: x=x'

```
-----
cc1=test1(x)
x''=g(x,y)
cc2=test2(x)
```

x'=f(x) (2)

```
-----
if cc1 goto T2
x=x''
cc2=test2(x)
```

x'=f(x) (2)

```
T2: if cc2 goto L1
exit: x live
```

Cycle 3

◆ loop:
 x'=f(x)
L1: x=x'
 cc1=test1(x)
 x''=g(x,y)
 cc2=test2(x)
 x'=f(x)(2)

 if cc1 goto T2
 x=x''
 cc2=test2(x)
 x'=f(x)(2)

T2:if ^cc2 goto exit
 goto L1

exit: x live

◆ loop:
 x'=f(x)
L1: x=x'
 cc1=test1(x)
 x''=g(x,y)
 cc2=test2(x)
 x'=f(x)

L2: if cc1 goto T3 (1)
 x=x''
 cc2=test2(x) (1)
 x'=f(x) (2)
 goto L3
T3:if ^cc2 goto exit (1)
 x=x'
 cc1=test1(x) (2)
 x''=g(x,y) (2)
 cc2=test2(x) (2)
 x'=f(x); goto L2 (3)

L3:if ^cc2 goto exit
 goto L1

◆ exit: x live

Cycle 4

- ◆ loop:

$x' = f(x)$

L1: $x = x'$

$cc1 = test1(x)$

$x'' = g(x, y)$

$cc2 = test2(x)$

$x' = f(x)$

L2: if $cc1$ goto T3 (1)

$x = x''$

$cc2 = test2(x)$ (1)

$x' = f(x)$ (2)

goto L3

T3: if $cc2$ goto exit (1)

$x = x'$

$cc1 = test1(x)$ (2)

$x'' = g(x, y)$ (2)

$cc2 = test2(x)$ (2)

$x' = f(x);$ goto L2 (3)

L3: if $cc2$ goto exit

goto L1

- ◆ exit: x live

L3: if $cc2$ goto exit (1)

$x = x'$

$cc1 = test1(x)$ (2)

$x'' = g(x, y)$ (2)

$cc2 = test2(x)$ (2)

$x' = f(x);$ goto L2 (3)

- ◆ 1 cycle/iteration if $test1(x)$ true:
L2->L2

- ◆ 2 cycles/iteration if $test1(x)$ false:
L2->L3->L2

Multiway Branching Hardware

- ◆ Each tree path represented by:
 - Target: the next tree instruction address for this path
 - Condition code mask: which cc's need to be T, which need to be F, and which are don't care, for this path to be taken
 - Execution mask: set of operations to execute if this path is taken
-

Multiway branching hardware

Fields for the example tree instruction

Target	Condition code mask	Execution mask
L3	cc1=F cc2=X	$x=x''$, cc2=test2(x''), $x'=f(x'')$
exit	cc1=T cc2=F	
L2	cc1=T cc2=T	$x=x''$, cc1=test1(x'), $x''=g(x',y)$, cc2=test2(x') $x'=f(x')$

Multiway branching hardware

- ◆ The upper bits of all tree targets are common and are sent to the icache at the beginning of cycle
 - ◆ The correct lower bits arrive as late select
 - ◆ All operations are executed, but those not on the taken path are not committed
 - ◆ Fast cycle time w/o branch stalls
-

Comparison to predication

- ◆ **Predication takes worst case dependence cycle:**
- ◆ **loop: $x=f(x)$
 $cc1=test1(x)$
 $x=(cc1 ? x : g(x,y))$
 $cc2=test2(x)$
if $cc2$ goto loop**
- ◆ **Best performance is 3 cycles/iteration**
- ◆ **Select (cond. move):**
- ◆ **loop: $x=f(x)$
 $cc1=test1(x)$
 $x'=g(x,y)$
 $x=(cc1 ? x : x')$
 $cc2=test2(x)$
if $cc2$ goto loop**
- ◆ **Best performance is 3 cycles/iteration**
- ◆ **Our approach: 1-2 cycles/iteration**

Memory System Challenges

- ◆ VLIW techniques improve only the infinite cache component
 - VLIW puts more demand on memory
 - ◆ Methods of tolerating cache latency
 - Hardware multithreading
 - Data touches for integer code
 - Code re-arrangement
 - Instruction touches
-

Ongoing Research

- ◆ Non-greedy enhanced pipeline scheduling (more register sensitive)
 - ◆ Profile directed feedback optimizations
 - ◆ Advanced interprocedural optimizations
 - ◆ Optimizations for technical workloads
 - ◆ Prefetching, memory system issues
 - ◆ Object code compatibility/OS issues
-