A Fast Interrupt Handling Scheme for VLIW Processors

E. Özer, S. W. Sathaye, K. N. Menezes, S. Banerjia, M. D. Jennings and T. M. Conte
Department of Electrical and Computer Engineering, North Carolina State University,
Raleigh,N.C., 27695-7911

Abstract

Interrupt handling in out-of-order execution processors requires complex hardware schemes to maintain the sequential state. The amount of hardware will be substantial in VLIW architectures due to the nature of issuing a very large number of instructions in each cycle. It is hard to implement precise interrupts in out-of-order execution machines, especially in VLIW processors. In this paper, we will apply the reorder buffer with future file and the history buffer methods to a VLIW platform, and present a novel scheme, called the ***Current-State Buffer****, which employs modest hardware with compiler support. Unlike the other interrupt handling schemes, the Current-State Buffer does not keep history state, result buffering or bypass mechanisms. It is a fast interrupt handling scheme with a relatively small buffer that records the execution and exception status of operations. It is suitable for embedded processors that require a fast interrupt handling mechanism with modest hardware.*

I. INTRODUCTION

Very Long Instruction Word (VLIW) architectures are statically scheduled architectures that contain multiple functional units (FUs). Multiple independent operations are sent to these FUs at each clock cycle. Independent operations that can be executed in parallel are determined at compile time, rather than at run time as in superscalar processors [5]. VLIW architectures [3] [9] are therefore classified as statically scheduled architectures. Commercial examples of VLIW systems include the Cydrome Cydra-5 [1] [9], and Multiflow TRACE [3] [7]. Embedded VLIW processors include Texas Instruments TMS320C62 [11], Philips TriMedia TM1000 [12] and Chromatic Mpact [15]. In VLIW processors, independent operations are grouped into a single long instruction in order to extract instruction level parallelism (ILP) [8]. These single long instructions are also referred to as **MultiOps**[16]**.** Because the order of operations is modified by the compiler, operations may not complete in sequential program order. At run-time, only the *scheduled program order*, which is the program order of MultiOps, is known by the processor.

In the case of an interrupt, the processor state is not consistent with the sequential state in out-of-order execution processors, including VLIW, because instructions are completed out-of-order. This is called the *interrupt problem* in out-of-order execution machines. Details on precise and imprecise interrupts can be found in [10]. Numerous solutions to the interrupt problem have been proposed for both scalar and superscalar processors. Several techniques to solve the interrupt problem for out-of-order execution processors have been proposed, such as the reorder buffer [10], the history buffer [10], the future file [10], checkpoint repair [4].

In this paper, we redefine the interrupt problem from the perspective of VLIW. The reorder buffer and history buffer schemes are revised and applied to general-purpose VLIW processors. A novel scheme called the *current-state buffer* is proposed as a fast interrupt handling solution for VLIW processors that are tailored to multimedia, communication and Digital Signal Processor (DSP) applications. The current-state buffer is a fast interrupt handling technique that employs modest hardware and compiler support. It is a simple buffer that records the execution and exception status information about the instructions without any result-buffering or history-state saving. It is based upon not re-executing the operations that have already completed execution before the interrupt occurs. Interrupts are detected and handled immediately after they occur. The current-state buffer requires some compiler scheduling support to maintain the correct processor state. The rest of the paper is organized as follows. Section II examines interrupt handling issues for VLIW processors. Section III introduces the current-state buffer scheme. Section IV

presents experimental results and Section V concludes the paper.

II. INTERRUPT HANDLING in a VLIW PROCESSOR

In a VLIW processor, interrupt handlers should not modify any of source registers in the interrupted operation because subsequent operations with respect to the original program order that use the same source register would need to be re-issued and re-executed after resumption. The processor cannot reexecute those operations because they may have already updated the processor state. Hence, it is assumed throughout the paper that interrupt handlers preserve registers in the interrupted processor state.

Debugging requires precise interrupt in any architecture model. Since the performance is less critical during debugging, the debugger can execute the original (unscheduled) code in program order.

In the following subsection, reorder buffer with future file and history buffer methods are extended for a VLIW processor, assuming a VLIW Less-than-or-Equals scheduling model [6].

Reorder Buffer with Future File

The reorder buffer keeps enough information about MultiOps and updates the processor state in scheduled program order. MultiOps in the scheduled code update the processor state sequentially, but the operations of the original program may not complete in original program order. The reorder buffer structure is shown in Figure 2.1.

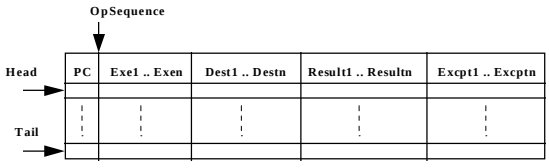


Figure 2.1 Reorder buffer.

The **PC** field is the program counter address of a MultiOp. $\mathbf{Exe_n}$ is a one bit location that is set when the nth operation is executed. $\mathbf{Dest_n}$ is the destination register number of the nth operation. $\mathbf{Result_n}$ is the result generated by the nth operation. $\mathbf{Excpt_n}$ is the exception conditions generated by the nth operation if one is generated. The reorder buffer is a circular buffer consisting of Head, Tail and OpSequence pointers. Head and Tail pointers point to the head and tail entries in the buffer, respectively. OpSequence pointer provides an index to the required operation field. The maximum buffer length is the longest latency operation plus one.

At MultiOp issue, the PC of the MultiOp is placed into the **PC** field of the reorder buffer entry pointed by tail pointer, and the destination register numbers are written into the **Dest** fields. When an operation executes without an exception, the result is written into the future file, its **Exe** bit is set and its result field is updated. The future file is a replica of the architectural register file. If an operation caused an exception, the exception status is recorded in the **Excpt** field. At each cycle, the entry pointed to by the head pointer is examined. The results are written into the architectural register file if all **Exe** bits of nonempty operation fields, i.e. excluding NOPs, are set in the entry. If an exception occurs in one of the operations, the whole MultiOp is said to be at the interrupt boundary. Then instruction issue is stopped, and all pipelines are flushed. The architectural register file loads its contents into the future file. The PC value of the MultiOp and the exception bits within the instruction are saved as part of the processor state to identify the source of the exception, and the reorder buffer contents are discarded. The first excepting operation, in left-to-right order in a MultiOp, will be reported if more than one operation in the MultiOp cause exceptions. A store buffer is required to buffer the writes by the store operations into the cache until they are removed from the reorder buffer.

The history buffer method can be extended in a similar way as the reorder buffer method for VLIW. The operating principle for the extended history buffer method is the same as the original history buffer scheme. It uses a history buffer to keep the old values of registers, rather than the latest values of registers as in the reorder buffer. The buffer structure is the same as the reorder buffer without a future file, except that the **Result** field is replaced with the **Old** field that retains the old value of a register.

The operation of the reorder buffer with future file and the history buffer is illustrated through an example. Consider the sample code in Figure 2.2, scheduled for a 2-issue VLIW machine with functional units as shown. The latencies of the **ADD, SUB** and **DIV** operations are 1, 1 and 2 cycles, respectively. This example illustrates the operation of both the reorder buffer with future file and the history buffer schemes since each buffer has similar operation mechanisms. The **DIV** operation in Mop2 is assumed to cause a trap, so it will not be re-executed after return from the interrupt handler. The execution steps are shown in Figure 2.3.

	ALU Unit	DIV Unit
Mop1	ADD (R2)	NOP
Mop2	ADD (R1)	DIV (R3)
Mop3	ADD (R4)	NOP
Mop4	SUB (R1)	NOP
Mop5	SUB (R4)	NOP

Figure 2.2 A sample code for a 2-issue VLIW processor.

In cycle 0, Mop1 is issued. In cycle 1, Mop2 is issued and the **ADD** operation in Mop1 is completed. In cycle 2, Mop3 is issued and the **ADD** operation in Mop2 is completed. In cycle 3, Mop4 is issued and the **ADD** in Mop3 is completed but the **DIV** in Mop2 excepted. It sets its Excpt bit but not its Exe bit. Afterwards, the trap handler is invoked. After return from the trap handler which consumes **n** cycles, Mop2 is re-issued in cycle (4+n). Note that the excepted operation no longer exists in the buffer due to the fact that the exception was a trap. In cycle (5+n), Mop3 is re-issued and the **ADD** operation in Mop2 completed. In cycle (6+n), Mop4 is re-issued and the **ADD** operation in Mop3 is completed. In cycle (7+n), Mop5 is issued and the **SUB** in Mop4 completed. The **SUB** in Mop5 completes in cycle (8+n).

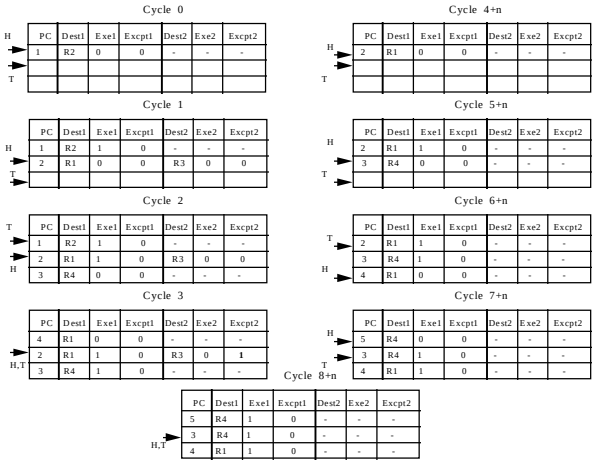


Figure 2.3 Execution steps in the reorder and the history buffers.

III. CURRENT-STATE BUFFER

The current-state buffer is a new interrupt handling scheme for a VLIW processor that signals interrupts immediately using a modest buffer. It supports both the Less-than-or-Equals and Equals scheduling models. It detects and signals interrupts as soon as they occur. The current-state buffer does not re-execute the operations that have already completed before the interrupt is taken when the program resumes. It relies on compiler scheduling support but requires only simple hardware. The hardware consists of a buffer, called current-state buffer, and a mask register that is shown in Figure 3.1. Each buffer entry consists of a **PC** field for each MultiOp, an **Exe** bit and **Excpt** bits for each operation.

The PC value of a MultiOp to be issued next is put into the **PC** field of the buffer entry pointed to by the tail pointer. Then the tail pointer is incremented. Each operation in the MultiOp is issued with an attached buffer address, an operation identifier, and exception tag. When an operation is executed, the associated buffer entry is accessed by the tags attached to each operation. Then its computed result is written into the register file, and its **Exe** bit is set in the current-state buffer if there is no exception. On each cycle, the entry at the head of the buffer is examined. The entry will be discarded by incrementing the head pointer if all **Exe** bits of nonempty operation fields are set. An exception will be signaled immediately to the exception logic if an operation causes an exception. First, the excepting operation's exception bits are set in the buffer, the issue is stopped and all pipelines are flushed. Then, the processor state, the relevant portion of memory and the current-state buffer contents between the head and tail pointers are saved. The saved buffer contents identify the source of the excepted operation.

After return from the interrupt handling routine, previously executed and completed operations are not re-executed. The interrupt handler routine returns by a special return instruction which switches to a special mode. In the special mode, the processor, the memory state and the current-state buffer contents are resumed, except the PC register. The buffer contents are examined starting from the head pointer. If all operations in a MultiOp completed execution, that MultiOp is not fetched. However, pipeline bubbles are inserted in order to honor dependence latencies of scheduled operations if there are MultiOps previously issued from the current-state buffer and being still executed in the pipelines. Otherwise, no pipeline bubbles are needed. If one or more operations in a MultiOp are still incomplete, the PC value of the entry is loaded into the PC register and the associated MultiOp is fetched and brought into the issue register. The **mask register** is loaded with the **Exe** bits of the entry, masking the issue register so that previously executed operations are not re-issued. This modified MultiOp is then issued to the functional units. This process continues until the head pointer passes the tail pointer. (This is detected by the buffer control logic.) As soon as this happens, the buffer control logic pops

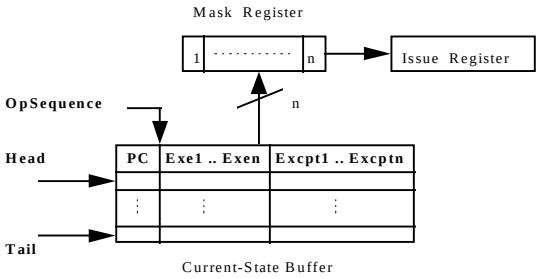


Figure 3.1 The current-state buffer and mask register.

up the PC register, either from the stack or a shadow PC register, and the processor resumes execution from the PC. In Figure 3.2, the operation of the current-state buffer is illustrated for the same sample code used in Figure 2.2.

The steps until the trap are identical to the previous illustration. After return from the interrupt handler in cycle (4+n), Mop2 and Mop3 do not need to be re-issued because operations were executed before the exception occurred. Since the excepted **DIV** operation is a trap, it is not executed either. Pipeline bubbles are not necessary since no MultiOp before Mop2 and Mop3 from the buffer is re-issued. So Mop4 is re-issued in cycle (4+n). In cycle (5+n), Mop5 is issued and the **SUB** in Mop4 completed. In cycle (6+n), the **SUB** in Mop5 completes. Hence, the current-state buffer requires less cycles to recover from an interrupt than the previous schemes.

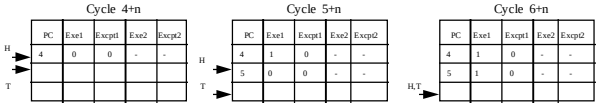


Figure 3.2 Execution steps in the current-state buffer.

Compiler Support

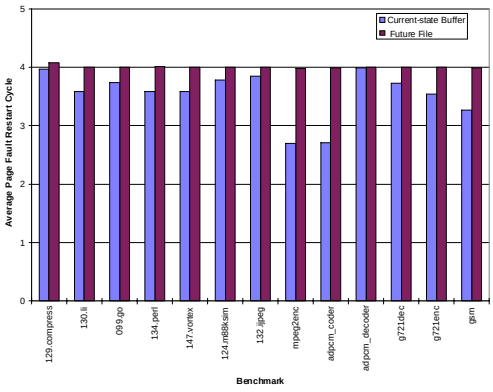
An anti-dependence between two operations may cause a problem in the current-state buffer when the *sink operation* of anti-dependence completes before the *source operation* of anti-dependence does. If the source operation excepts, the interrupt handler may read an incorrect value of the source register on which the anti-dependence occurs. Such an anti-dependence is called an *unsafe* anti-dependence. The compiler can treat unsafe anti-dependences between operations as if they are flow dependences. It must guarantee that the destination register of the sink operation is not modified until the source operation completes. Only *unsafe anti-dependences* are modified by the compiler. Safe anti-dependences remain in the code. In the post-pass scheduling phase of the compiler, unsafe anti-dependences are converted into flow-dependences. However, there is a drawback in treating unsafe anti-dependences as flow-dependences. The scheduling times tend to increase because of NOPs that have to be inserted to increase the distance between two anti-dependent operations. This could increase the schedule length and degrade the performance, particularly in Equals scheduling model because it is likely that Equals model produces more unsafe anti-dependences than Less-than-or-Equals model. The impact will be empirically studied for the Less-than-or-Equals scheduling model in Section IV.

IV. EXPERIMENTAL RESULTS

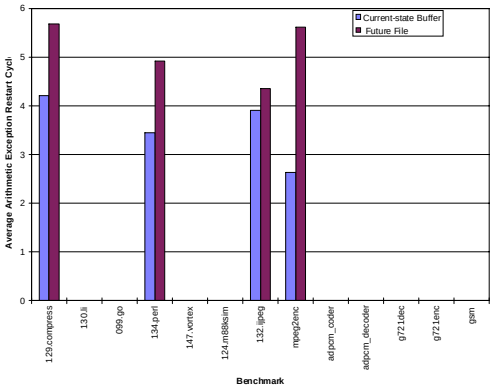
The architectural framework used in the experiments is an 8-issue VLIW machine with the Less-than-or-Equals scheduling model. Experiments were performed to measure the interrupt handling restart cycles and interrupt detection delay cycles for each scheme. Also, slowdown in the machine using the current-state buffer with no unsafe anti-dependences in the code is measured with respect to the same machine using the current-state buffer with unsafe anti-dependences. The history buffer scheme achieves the same performance as the reorder buffer with future file. Therefore, only the reorder buffer with future file scheme is compared against the current-state buffer in the experimental results. There are three integer, two floating-point, two memory and one branch functional units in the model architecture. A perfect cache is assumed. Benchmarks from SPECint95 [14] and MediaBench [13] are evaluated in the experiments. SPECint95 contains general-purpose integer programs. Programs from MediaBench are embedded multimedia and communication applications. Traces are generated by using the IMPACT compiler [2] and are fed into a trace-driven simulator. Experiments are done for 100 million instructions. The simulator generates interrupts on potentially excepting instructions (PEI) on a random basis and measures the restart cycles and the interrupt detection delay cycles per interrupt for each scheme. The simulator generates a large number of interrupts to gather more accurate statistics about interrupts. Arithmetic and page fault exceptions are analyzed separately to see how each scheme reacts to different types of exceptions. Then the simulator computes restart and exception detection cycles for each exception type and calculates the average restart and exception detection cycles for each benchmark program.

The IMPACT compiler is modified in such a way that any unsafe anti-dependences are treated like flow-dependences. The modified compiler generates

traces for the current-state buffer. The simulations are performed assuming 32 integer and 32 floating-point registers. Infinite store buffer is used to prevent any instruction issue stall cycles. Figure 4.1a shows the average restart cycles for page fault exceptions in each scheme and Figure 4.1b shows the average restart cycles for arithmetic exceptions. Average restart cycles are overhead cycles per exception for re-executing some operations after the processor resumes execution.



(a)

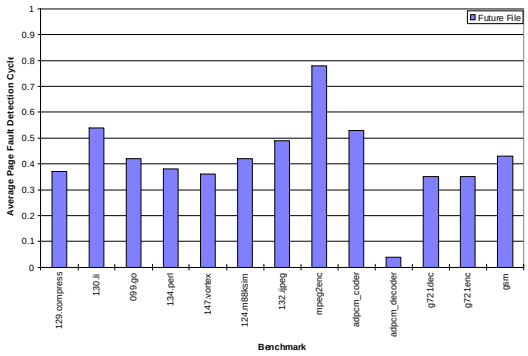


(b)

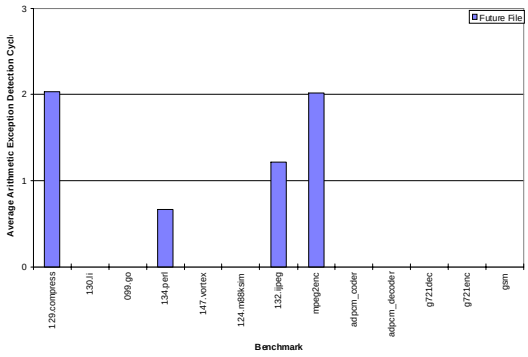
Figure 4.1. Average page fault and arithmetic exception restart cycles for the current state buffer and the future file.

Load and store operations cause page faults and floating-point operations cause arithmetic exceptions. The current-state buffer outperforms the future file for both types of interrupts in all benchmarks, particularly for arithmetic exceptions. Since the floating-point operations are long latency operations, the current-state buffer does not re-execute later short latency operations that have already completed. So, the average number of restart cycles for arithmetic exceptions are relatively small as compared to the future file. The benchmarks *130.li, 099.go, 147.vortex, 124.m88ksim, adpcm_coder, adpcm_decoder, g721dec, g721enc* and *gsm* generate almost no arithmetic exceptions as seen in Figure 4.1b.

In Figure 4.2, average exception detection delay cycles are shown for the reorder buffer with future file. Average exception detection delay is the delay between detecting an interrupt and signaling it per interrupt. Since the current-state buffer detects and signals the interrupts immediately, its interrupt detection delay is always zero.



(a)



(b)

Figure 4.2 Average page fault and arithmetic exception detection cycles for the reorder buffer with future file.

So far, we have shown the performance gain by the current-state buffer. However, the scheme requires a different scheduling technique, treating all unsafe anti-dependences like flow-dependences. This makes the machine slow with respect to the machine allowing all unsafe anti-dependences. The scheduler inserts NOPs to increase the dependence distance. Figure 4.3 shows slowdown in percentage in the machine for the current-state buffer without unsafe anti-dependences versus current-state buffer with unsafe anti-dependences. The benchmarks *130.li, 124.m88ksim, mpeg2enc, adpcm_coder* and *adpcm_decoder* do not experience slowdown. *099.go*

has the highest slowdown 0.6% due to a higher number of unsafe anti dependences. Slowdown is very modest, even in the worst-case. Benchmarks from MediaBench have almost no slowdown (*gsm* is the highest at 0.01%).

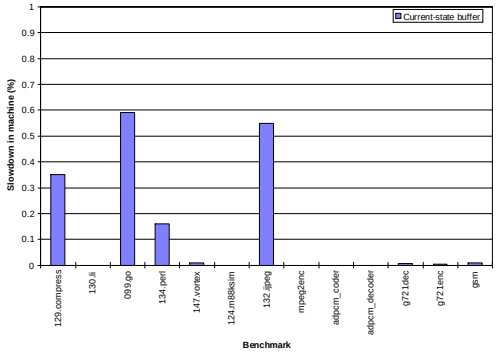


Figure 4.3. Percentage slowdown in the machine using the current-state buffer. No slowdown exceeds 0.6%.

V. CONCLUDING REMARKS

In this paper, we revise the reorder buffer with future file and history buffer schemes for VLIW processors and propose the current-state buffer, a new scheme for interrupt-handling for VLIW architectures. The current-state buffer monitors and records the execution and exception status of operations. It detects and handles interrupts immediately. Unlike many other interrupt-handling schemes, such as the reorder and history buffers, the current-state buffer does not require history state saving or result buffering. The processor does not re-execute operations that have already completed before the interrupt occurred. However, the current-state buffer scheme requires compiler intervention in the case of unsafe anti-dependences between operations. Anti-dependences are treated like flow-dependences by the compiler during the post-pass scheduling phase to eliminate this hazard.

Our experimental results show that the current-state buffer incurs a small number of restart cycles per interrupt-basis compared to the reorder buffer with future file and the history buffer methods. Furthermore, the current-state buffer provides a zero-cycle interrupt detection mechanism. Slowdown caused by treating anti-dependences like flow-dependences is relatively small, especially in embedded applications. Consequently, the current-state buffer, because of modest hardware and compiler support, is appropriate for embedded VLIW processors.

Acknowledgments

This work was supported by grants from IBM, Intel and HP.

REFERENCES

[1] G.R. Beck, D.W.L. Yen, T.L. Anderson," The Cydra 5 Minisupercomputer: Architecture and Implementation," *The Journal of Supercomputing*,7, 1993.

[2] P.P. Chang, S.A. Mahlke, W.Y. Chen, N.J. Warter, and W. W. Hwu, "IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors," *Proc. 18th Ann. Int'l. Symp. Computer Architecture*, May 1991.

[3] R.P. Colwell, R.P. Nix, J.J. O'Donnell, D.B. Popworth, and P.K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler,"*IEEE Trans. on Comp.*,vol.37,No.8, August 1988.

[4] W.W. Hwu, and Y. N. Patt, " Checkpoint Repair for High-Performance Out-of-Order Execution Machines", *IEEE Trans. on Comp.*,Vol.C-36, No12, Dec. 1987.

[5] M. Johnson, *Superscalar Microprocessor Design*, Englewood Cliffs, NJ: Prentice-Hall, 1991

[6] V. Kathail, M. Schlansker, and B. R. Rau, " HPL Playdoh Architecture Specification: Version 1.0," *Hewlett-Packard Computer Systems Laboratory*, HPL-93-80, Feb. 1994.

[7] P. G. Lowney, S. M. Freundenberger, T. J. Karzes, W.D. Lichtenstein, R. P. Nix, J.S. O'Donnell, and J. C. Ruttenberg, " The Multiflow Trace Scheduling Compiler", *The Journal of Supercomputing*, 7, 1993.

[8] B.R. Rau, J.A. Fisher, "Instruction-level Parallel Processing: History, Overview and Perspective," *Hewlett-Packard Computer Systems Laboratory*, HPL-92-132, Oct. 1992.

[9] B.R. Rau, D.W.L. Yen, W. Yen, and R. A. Towle," The Cydra 5 Departmental Supercomputer," *IEEE Computer*, January 1989.

[10] J.E. Smith, and A.R. Pleszkun," Implementing Precise Interrupts in Pipelined Processors," *IEEE Trans. on Comp.*, Vol.37, May 1988.

[11] *TMS320C62XX CPU and Instruction Set Reference Guide*, Texas Instruments, July 1997.

[12] *TM 1000 Preliminary Data Book*, Philips Electronics North America Coporation, 1997.

[13] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems", *Proc. 30th Ann. Int'l Symp. Microarchitecture (MICRO 30)*, 1997.

[14] K. Dixit and J. Reilly, "SPEC95 Questions and Answers", *SPEC Newsletters*, Sept. 1995.

[15] P. Kalapathy, "Hardware/Software Interactions on the Mpact," *IEEE Micro*, Mar./Apr. 1997.

[16] B.R. Rau, "Dynamically Scheduled VLIW Processors," *Proc. 26th Ann. Int'l. Symp. Microarchitecture*, Dec. 1993.