Zero-Cycle Loads: Microarchitecture Support for Reducing Load Latency

Todd M. Austin   Gurindar S. Sohi

University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706
{austin,sohi}@cs.wisc.edu

Abstract

Untolerated load instruction latencies often have a significant impact on overall program performance. As one means of mitigating this effect, we present an aggressive hardware-based mechanism that provides effective support for reducing the latency of load instructions.

Through the judicious use of instruction predecode, base register caching, and fast address calculation, it becomes possible to complete load instructions up to two cycles earlier than traditional pipeline designs. For a pipeline with one cycle data cache access, this results in what we term a zero-cycle load. A zero-cycle load produces a result prior to reaching the execute stage of the pipeline, allowing subsequent dependent instructions to issue unfettered by load dependencies. Programs executing on processors with support for zero-cycle loads experience significantly fewer pipeline stalls due to load instructions and increased overall performance.

We present two pipeline designs supporting zero-cycle loads: one for pipelines with a single stage of instruction decode, and another for pipelines with multiple decode stages. We evaluate these designs in a number of contexts: with and without software support, in-order vs. out-of-order issue, and on architectures with many and few registers. We find that our approach is quite effective at reducing the impact of load latency, even more so on architectures with in-order issue and few registers.

1 Introduction

High-performance computing requires high sustained instruction issue rates, a goal which can only be achieved if pipeline hazards are minimized. Data hazards, an impediment to performance caused by instructions stalling for results from executing instructions, can be mitigated by tolerating or reducing instruction execution latencies.

For many programs, the dominating source of data hazards can be attributed to load instructions. These operations occur frequently and have longer latency than most other non-numeric instructions because they combine address calculation, data cache access, and occasional accesses to lower levels of the data memory hierarchy in a single instruction.

A significant body of work is dedicated to reducing the impact of load latency. The techniques can be broadly bisected into two approaches: latency tolerating techniques and latency reducing techniques. Tolerating techniques require that independent operations be moved into unused pipeline delay slots. This reallocation of processor resources can be performed either at compile time using instruction scheduling or at run time using techniques such as out-of-order issue, non-blocking loads, or multi-threading. Reducing techniques decrease or eliminate some component of load instruction latency; for example, register allocation eliminates the entire load operation, or (loop) blocking eliminates many cache miss latencies.

In this paper, we present a microarchitecture design capable of reducing the latency of load instructions. Through the application of instruction predecode, base register caching, and fast address calculation, it becomes possible to complete load instructions up to two cycles earlier than traditional pipeline designs. For a pipeline with one cycle data cache access, this results in what we term a *zero-cycle load*. A zero-cycle load produces a result prior to reaching the execute stage of the pipeline, allowing subsequent dependent instructions to issue unencumbered by load instruction hazards. Programs executing on processors with support for zero-cycle loads experience significantly fewer pipeline stalls due to load instructions and increased overall performance.

We present two pipeline designs supporting zero-cycle loads: an aggressive design for pipelines with a single stage of instruction decode, and a less aggressive design for pipelines with multiple decode stages. We evaluate these designs in a number of contexts: with and without software support, in-order vs. out-of-order issue, and on architectures with many and few registers. We find that our approach is quite effective at reducing the impact of load latency, even more so on architectures with in-order issue and few registers.

The remainder of this paper is organized as follows: Section 2 details the microarchitecture support required to implement zero-cycle loads. Section 3 presents a detailed example of zero-cycle loads in action. In Section 4, we present results of simulation-based performance studies and Section 5 describes related work. Finally, Section 6 presents a summary and conclusions.

2 Zero-Cycle Loads

A load, while being a single instruction, is composed of several smaller component operations, many of which must occur in a specific order prior to completion of the load and delivery of a value from memory. Figure 1a illustrates the major component operations of a load and their required order. [1]

As shown in Figure 1b, a traditional pipeline fetches loads in the IF stage of the pipeline. Identifying, aligning, and reading the register file occurs in the ID stage of the pipeline. In designs with very fast clocks and wide issue, these operations are often split

[1] Many variations exist upon this basic template; for example, some pipelines require address translation to complete before accessing the data cache, other designs require that register file access occur after a load has been aligned into a pipe that services load instructions. This basic template is, however, representative of many modern pipeline designs.

Appears in: "Proceedings of the 28th Annual International Symposium on Microarchitecture"

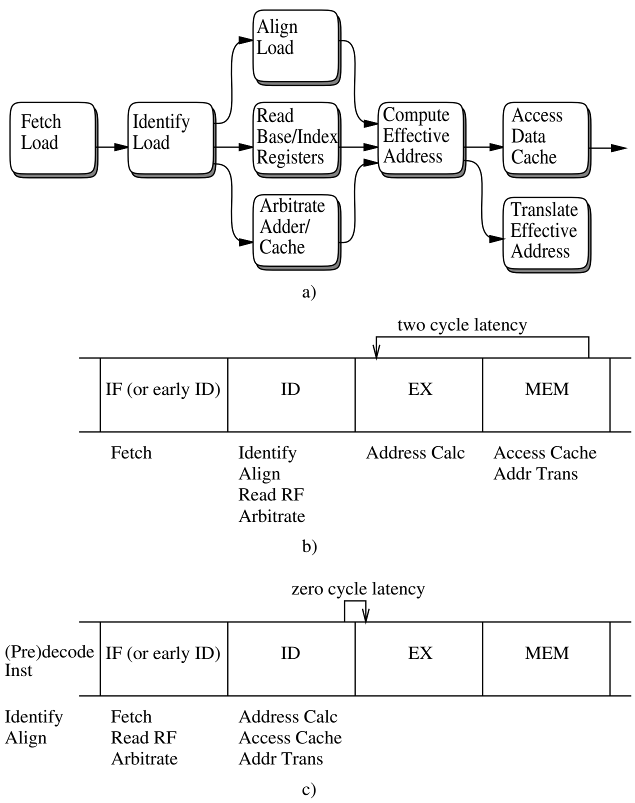

Figure 1: Zero-Cycle Loads.

across multiple decode stages. Effective address generation occurs in the EX stage of the pipeline, and data cache access and address translation in the MEM stage of the pipeline. This organization creates a two cycle latency for load instructions.

With support for zero-cycle loads (Figure 1c), loads can complete up to two cycles earlier than the traditional pipeline. We accomplish this optimization with two basic mechanisms. First, we use instruction predecode and base register caching to reduce the time required to decode and issue a load instruction. Instruction predecode reduces the latency to identify and align the loads in a group of fetched instructions. Base register caching provides the necessary high-bandwidth access to base and index register values early in the pipeline. Second, we employ *fast address calculation* [APS95] to reduce the latency of data cache access. Fast address calculation is a stateless set index predictor that allows address calculation and data cache access to proceed in parallel.

The combination of early issue and faster data cache access results in a pipeline design capable producing a load result two cycles earlier than traditional organizations. For pipelines with a single cycle data cache access, it becomes possible to forward a load result into the execute stage of the pipeline. If latency is defined as the number of cycles from the beginning of the execute stage for an operation to produce a result, successfully speculated loads will appear to have zero latency – hence the moniker "zero-cycle loads".

Not all loads can execute with zero latency. Early issue introduces new register and memory interlocks, and fast address calculation will occasionally mispredict the effective address, necessitating a recovery mechanism. In the following subsections, we more fully explore the implementation of zero-cycle loads, examining both the organizational and pipeline control impacts of its use. Two designs are presented in detail: an aggressive design for pipelines with only a single stage of instruction decode, and a less aggressive design for pipelines with multiple decode stages.

2.1 Implementation with One Decode Stage

Achieving a zero-cycle load in a five stage pipeline is a very challenging task – the load instruction must complete in only two pipeline stages. Assuming data cache access takes one cycle, all preceeding component operations must complete in only a single cycle. Figure 2 shows one approach to implementing zero-cycle loads in a pipeline with a single decode stage.

2.1.1 Organization

Fetch Stage In the fetch stage of the processor, the instruction cache and *base register and index cache* (or BRIC) are accessed in parallel with the address of the current PC.

The instruction cache returns both instructions and predecode information. The predecode information is generated at instruction cache misses and describes the loads contained in the fetched instructions. The predecode data is supplied directly to the pipes which execute loads, permitting the tasks of fetching, identifying, and aligning loads to complete by the end of the fetch stage.

The predecode data for each load consists of three fields: the addressing mode, base register type, and offset. The addressing mode field specifies either a `register+constant` or `register+register` addressing (if supported in the ISA). The base register type is one of the following: SP load, GP load, or other load. *SP load* and *GP load* specifies a load using the stack or global pointer [CCH+87] as a base register, respectively. *Other load* specifies a load using a register other than the stack or global pointer as a base register. The offset field specifies the offset of the load if it is contained as an immediate value in the instruction.

The BRIC is a small cache indexed by the address of a load, producing a register pair: the base register value and the index register value (unused if a `register+constant` mode load). During execution, the BRIC builds an association between the address of loads executed and their base and index register values. This address-to-register value association allows register access to complete by the end of the fetch stage of the pipeline. If the BRIC misses, an entry is replaced after the base and index register values have been read from the integer register file.

Loads that use the stack or global pointers are executed quite frequently [APS95] – we can increase the effective capacity of the BRIC by using an alternate means to supply these loads with a base register value. As shown in Figure 2, two registers are used to cache the global and stack pointer values. When an access is made, the type field of the predecode data is used to select the correct base register source.

Any cached register value must be updated whenever the corresponding register file value is updated. Since multiple loads may be using the same base and index registers, a value written into the BRIC may have to be stored into multiple locations. Consequently, the BRIC is a complex memory structure, supporting multiple access ports and multi-cast writes. We do not expect it to be very large before its access time impacts processor cycle time, hence, we consider only very small sizes – on the order of 4-64 elements. We also consider organizations without a BRIC, a configuration particularly useful to programs with a high frequency of global and stack pointer accesses.

Decode Stage In the decode stage of the pipeline, the base register and offset pair produced in the fetch stage are combined using fast address calculation. (The fast address calculation mechanism is represented by the box labeled FAC in Figure 2.) Fast address

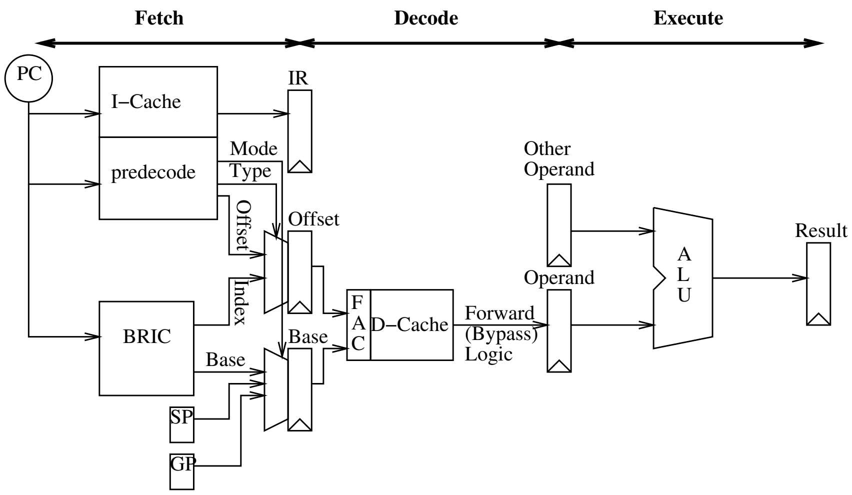

Figure 2: Pipelined Implementation with One Decode Stage.

calculation is a fast stateless set index predictor that leverages off the organization of on-chip data cache to allow non-speculative address calculation to proceed in parallel with data cache access.

On-chip caches are organized as wide two-dimensional arrays of memory cells (as shown in Figure 3). This geometry minimizes access time by reducing wire lengths. Each row of a cache array typically contains one or more data blocks [WRP92, WJ94]. To access a word in the cache, the set index portion of the effective address is used to read an entire cache row from the data array and a tag value from the tag array. Late in the cache access cycle, a multiplexor circuit uses the block offset part of the effective address to select the referenced word from the cache row. At approximately the same time, the tag portion of the effective address is compared to the tag value from the tag array to determine if the access hit in the cache. Consequently, on-chip cache organizations require the set index portion of the effective address at the beginning of the cache access cycle and the block offset and tag portion later, after the cache row and tag have been read.

As shown in Figure 3, the set index portion of the effective address is supplied very early in the cache access cycle by OR'ing the set index portion of the base and offset. We call this limited form of addition *carry-free* addition as this operation ignores any carries that may have been generated in or propagated into the set index portion of the address calculation. Because many offsets are small [APS95], the set index portion of the offset will often be zero, allowing this fast computation to succeed. For larger offsets, like those applied to the global or stack pointer, we can use software support to align pointer values, thereby increasing the likelihood that the set index portion of the base register value is zero.

In parallel with access of the cache data and tag arrays, full adders compute the block offset and tag portion of the effective address. Later in the cache access cycle, the block offset selects the correct word from the cache row, and the tag is compared to the tag value read from the tag array. If the effective address prediction succeeds, the value read from the data cache is available by the end of the decode stage of the pipeline. It may then be forwarded, via bypass data paths, to a dependent instruction.

The pipeline organization in Figure 2 requires two new data paths (per pipe). A path must be added to allow forwarding of register values from the BRIC directly to the data cache. (This path does not normally exist on traditional pipeline organizations

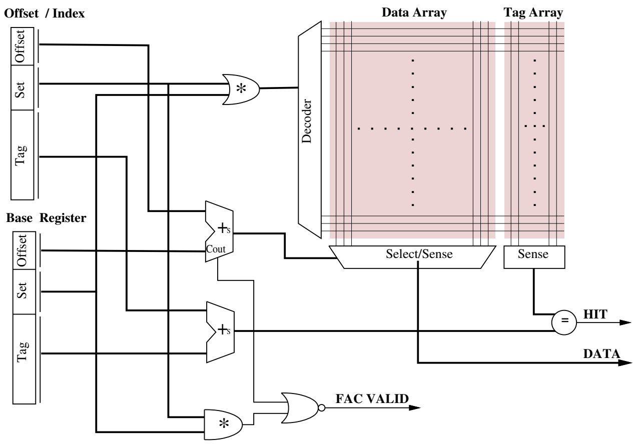

Figure 3: Fast Address Calculation. Bold lines indicate a bus, gates with an '*' signify a replicated gate for every line of the connected bus.

as all values from the fetch/decode stages of the pipeline will first pass through the execute stage before arriving at the data cache ports.) In addition, the data path used to write back register values to the register file must also be extended to supply register values to the BRIC. All other data paths used to facilitate zero-cycle loads (i.e., D-cache to ALU, ALU to D-cache, ALU to ALU, D-cache to D-cache) already exist in traditional pipeline organizations.

2.1.2 Pipeline Control

As with most pipeline optimizations, the brunt of the complexity is placed on the pipeline control circuitry. The following logic equation summarizes the condition under which a zero-cycle load will succeed:

$$\begin{aligned} ZCL\_Valid \;\leftarrow\; & BRIC\_Hit \wedge FAC\_Valid \\ & \wedge\, Port\_Allocated \wedge DCache\_Hit \\ & \wedge\, \overline{Reg\_Interlock} \wedge \overline{Mem\_Interlock} \end{aligned}$$

$BRIC\_Hit$ indicates if the load address hit in the BRIC.

FAC_Valid indicates if fast address calculation succeeded. Fast address calculation succeeds when no carries are propagated

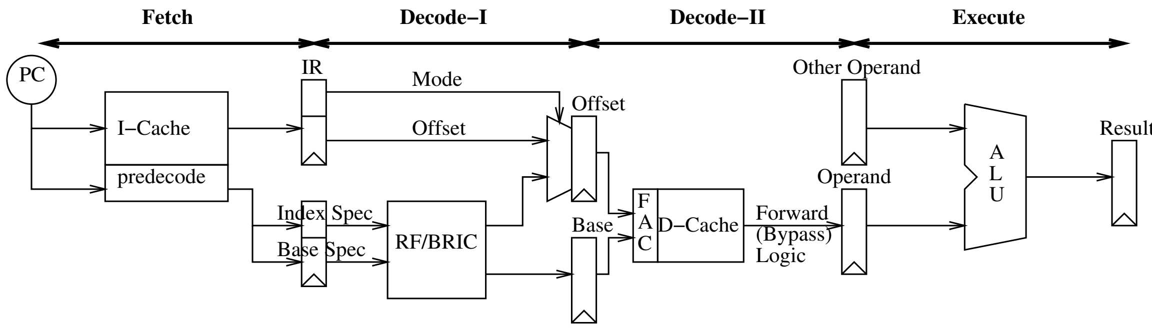

Figure 4: Pipelined Implementation with Multiple Decode Stages.

into or generated in the set index part of the effective address computation. (The verification circuit is shown in the lower portion of Figure 3.)

$Port\_Allocated$ indicates if a data cache port is available for the speculative load. This signal is required because accessing the data cache from multiple pipeline stages provides more points of access to the data cache than ports available, necessitating port allocation on a per cycle basis.

$DCache\_Hit$ indicates if the load hit in the data cache.

$Reg\_Interlock$ indicates whether a data hazard exists between the base and index register values used by the zero-cycle load and the register results of programmatically earlier instructions still executing or waiting to execute.

$Mem\_Interlock$ is analogous to $Reg\_Interlock$, but detects conflicts through memory. An interlock through memory occurs whenever an earlier store instruction with a matching effective address (or an unknown effective address) has not completed execution.

If a zero-cycle load is not possible or fails due to a mispredicted effective address, there are a number of options available for recovery. If the BRIC misses, the register values read in the decode stage of the pipeline can be used to re-execute the access using fast address calculation in the execute stage of the pipeline. If successful, the load will complete in one cycle. If fast address calculation fails, a non-speculative effective address can be computed in the execute stage of the pipeline, with subsequent data cache access in the memory stage. Alternatively, if an adder is available for use in the decode stage of the pipeline, non-speculative effective address could be performed on the register values from the BRIC, and the failed access could be re-executed in the execute stage of the processor. If an interlock condition exists, the load must stall until it clears, at which point the access can proceed, possibly employing fast address calculation if the interlock condition clears before address generation completes. In the worse case, the BRIC will miss, forcing re-execution in the execute stage, where fast address calculation will fail, resulting in re-execution in the memory stage of the processor – a worse case latency of two cycles (given sufficient data cache bandwidth).

In some designs, it may be possible to detect a failure condition early enough to prevent a speculative access. This pipeline will benefit from less wasted data cache bandwidth. For the one decode stage design, we assume a BRIC miss or failure to arbitrate a data cache port are the only conditions that can elide a data cache access. We have been intentionally conservative in deciding what conditions may elide a speculative data cache accesses. This strategy ensures that our failure detection logic is simple and fast, which minimizes impacts on the pipeline control critical path and processor cycle time.

2.2 Implementation with Multiple Decode Stages

The increased complexity of instruction decode created by wider issue widths and faster clock speeds has forced many recent designs to increase the number of pipeline stages between instruction fetch and execute. (Stages which we collectively call decode stages.) For example, the DEC 21164 [Gwe94a] has three decode stages and the MIPS R10000 [Gwe94b] has two. Adding more decode stages increases the mispredicted branch penalty, however, architects have compensated for this penalty by increasing branch prediction accuracy through such means as larger branch target buffers or more effective predictors, *e.g.*, two-level adaptive.

Given extra decode stages, the task of implementing zero-cycle loads becomes markedly easier. Figure 4 shows one approach to providing support for zero-cycle loads on a pipeline with two decode stages.

Register access is delayed to the first decode stage of the pipeline. This modification eliminates the need for a complex address-indexed BRIC in the fetch stage, permitting direct register file access using register specifiers. In Figure 4, we've arbitrarily assumed that instructions are not fully decoded until the end of the first decode stage, thus the base and index register specifiers are supplied by instruction predecode. In some designs, it may be possible to decode the register specifiers and access the register file in a single cycle, eliminating the need for instruction predecode completely.

In some pipelines, it may not be possible to access the integer register file in the first decode stage without supplying more ports, which greatly increases the risk of impacting processor cycle time. A better alternative for these designs may be to adapt the BRIC as a means for caching register values. We and others [FP91] have found a significant amount of temporal locality in base and index register accesses. A small cache, on the order of 4 to 8 entries provides the necessary bandwidth to register values without increasing the number of ports on the existing integer register file. Like the original BRIC, a miss initiates a replacement which is available for use after the base and index registers have been read from the integer register file. However, unlike the original BRIC, this storage need not support multi-cast writes, since any register value will reside in at most one cell.

The extra decode stage makes it possible to detect more interlock conditions prior to speculative data cache access. For this design, we assume register interlock conditions are detected early enough to terminate the speculative access. As a result, a failure in fast address calculation, a memory interlock condition, or a data cache miss are the only signals that do not provide early termination of a speculative access. These conditions cannot be detected early because testing for them requires values that are only available after the start of the data cache access cycle.

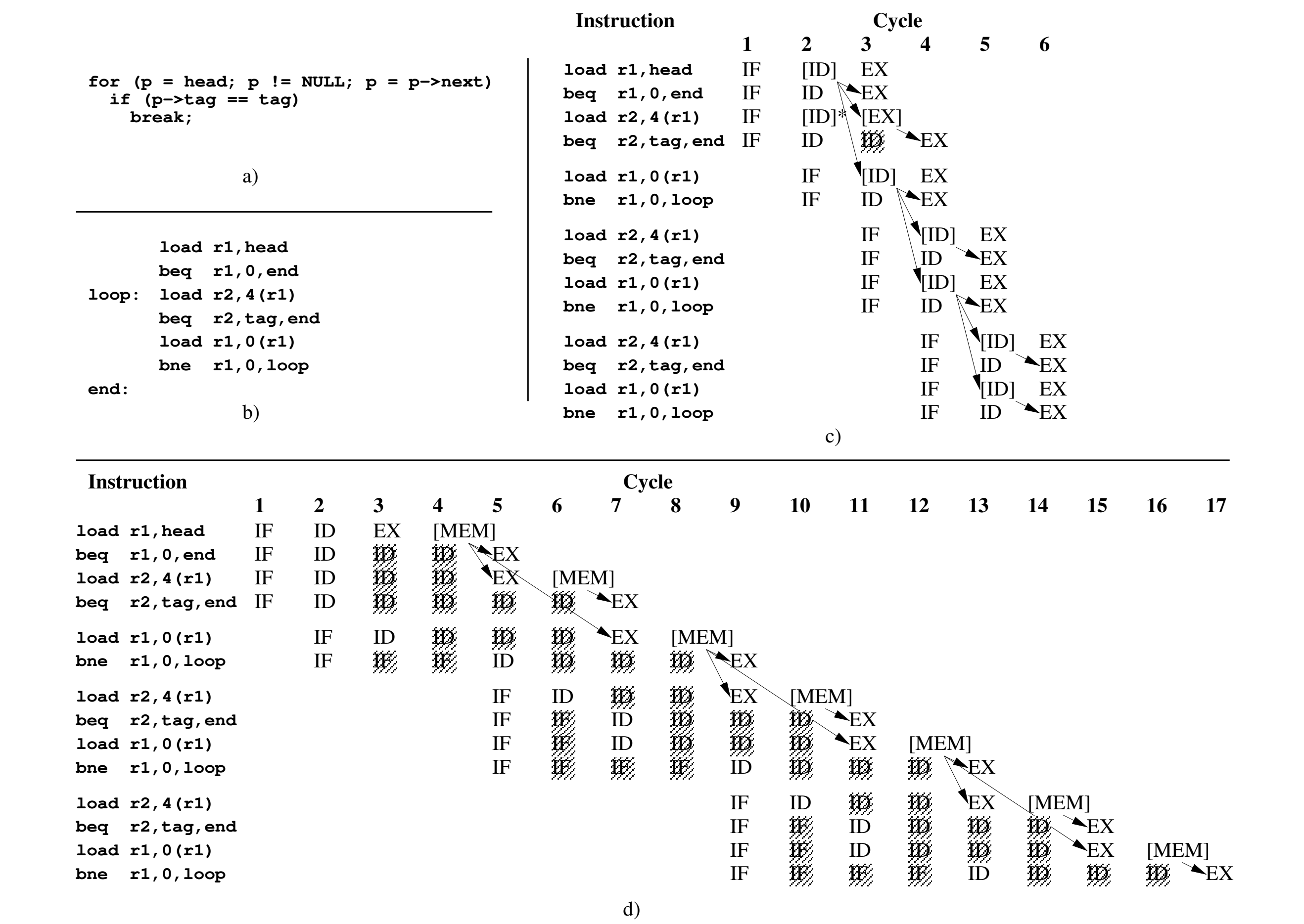

Figure 5: Pointer Chasing Example with and without Zero-Cycle Loads.

By the end of the first decode stage, this design and the single decode stage design converge. In the second decode stage, fast address calculation is used to compute the effective address and the data cache is accessed.

2.3 Further Design Considerations

We found in a earlier report [APS95] that fast address calculation for `register+register` mode accesses fails often. Use of this addressing mode can be directly attributed to codes where strength reduction of array subscript expressions [ASU86] is not possible or fails, resulting in many large index variable offsets. We can improve the design by not speculating loads using this addressing mode. We instead compute the effective address during the decode stage of the pipeline and access the data cache in the execute stage. As a result, a `register+register` mode access that hits in the BRIC will complete in only one cycle, two cycles otherwise.

Other failure conditions may manifest due to speculative data cache access. If a fault does occur, *e.g.*, access to an invalid page table entry, the fault must be masked until the instruction becomes non-speculative. Once the speculative access is verified as correct, posting the fault proceeds as in the non-speculative case.

Our fast address generation mechanism assumes that data cache access can start as soon as the set index part of the effective address is available. If this is not the case, *e.g.*, the cache is indexed with a translated physical address or cache bank selection uses part of the block offset, fast address calculation can not be used. (Our early issue mechanism, however, can still be applied.)

3 A Working Example

Figure 5 illustrates the performance advantage of zero-cycle loads. Figure 5a shows a simple C code fragment which traverses a linked list searching for an element with a matching tag field (often referred to as "pointer chasing"). Figure 5b shows the assembly output for a MIPS-like target (without architected delay slots) as produced by the GNU GCC compiler. This code sequence was selected because it is a very common idiom in C codes, and it is difficult to tolerate the latency of the loads with compile-time scheduling. Moving either load in the loop would require a global scheduling technique because both are preceded by branches. In addition, moving the first load into a previous iteration of the loop would require support for masking faults since a NULL pointer may be dereferenced.

Figure 5c and 5d depict the code executing on a 4-way in-order issue superscalar processor with and without support for zero-cycle loads, respectively. In both executions, the example assumes perfect branch prediction, one cycle data cache access latency, and unlimited functional unit resources. The stage specifiers contained within brackets, *e.g.*, [ID], denote a data cache access occurred during that cycle. Arrows indicate where values from memory were forwarded to other instructions. The shaded stage specifiers indicate that the instruction was stalled in that stage for one cycle. In the execution with support for zero-cycle loads, all BRIC accesses hit and all fast address calculations succeed. Incorrectly speculated accesses are denoted with an asterisk.

As seen by comparing the two execution examples, support for zero-cycle loads significantly reduces the number of cycles to execute the code sequence. Without zero-cycle load support, each iteration requires four cycles to execute; with zero-cycle load support, each iteration requires only a single cycle to execute, as both load results are available by the time the two branches reach the execute stage of the processor. Reducing the latency of the load instructions eliminates nearly all stalls. Only one access is misspeculated (marked with an asterisk). This access must be re-executed in the execute stage of the processor because the earlier load issued in the same cycle created a value used by the later load, a violation of a RAW dependence. (This failure condition would be indicated by the signal $Reg\_Interlock$.)

Although not shown in the figure, running the same code on a 4-way out-of-order issue processor without zero-cycle load support requires two cycles to execute each iteration. The out-of-order issue processor cannot achieve one iteration per cycle because the code segment contains a recurrence requiring two cycles per iteration (one cycle for address calculation followed by one cycle to access the data cache). On the same out-of-order issue processor with support for zero-cycle loads, the latency reduction capability of fast address calculation allows each iteration of the recurrence to complete in one cycle.

4 Experimental Evaluation

We evaluated the impact of zero-cycle loads by extending a detailed timing simulator to support zero-cycle loads and examining the performance of programs running on the extended simulator. We varied the processor issue model, level of software support, and number of architected registers to see what effects these changes had on the efficacy of zero-cycle loads.

4.1 Experimental Framework

4.1.1 Compiler Tools and Software Support

All programs were compiled with GNU GCC (version 2.6.0), GNU GAS (version 2.2), and GNU GLD (version 2.3) with maximum optimization (-O3) and loop unrolling enabled (-funroll-loops). The Fortran codes were first converted to C using AT&T F2C version 1994.09.27.

We added software support to GCC and GLD to improve the prediction performance of fast address calculation. A complete description of our optimizations as well as the parameters used when compiling the codes can be found in [APS95]. In short, our software support works to align pointers and reduce the size of load offsets, two transformations which improve the prediction accuracy of fast address calculation. It is important to note that these optimizations are not required, and their implementation is quite simple, totalling less than 1000 lines of C code.

4.1.2 Simulation Tools

All experiments were performed on an extended (virtual) MIPS-like architecture. The architecture implements a superset of the MIPS-I instruction set [KH92], with the following extensions:

- extended addressing modes: `register+register` and post-increment and decrement are included
- no architected delay slots

Our baseline simulator is detailed in Table 1. The simulator executes only user-level instructions, performing a detailed timing simulation of 4-way superscalar microprocessor and the first level of instruction and data cache memory.

Fetch Width	4 instructions
Fetch Interface	able to fetch any 4 contiguous instructions in the same cache block per cycle
I-cache	16k direct-mapped, 32 byte blocks, 6 cycle miss latency
Branch Predictor	2048 entry direct-mapped BTB with 2-bit saturating counters, 2 or 3 cycle misprediction penalty
In-Order Issue Mechanism	in-order issue of up to 4 operations per cycle, allows out-of-order completion
Out-of-Order Issue Mechanism	out-of-order issue of up to 4 operations per cycle, 16 entry register update unit, 8 entry store queue, loads may execute when all prior store addresses are known
Functional Units	4-integer ALU, 2-load/store units, 2-FP adders, 1-integer MULT/DIV, 1-FP MULT/DIV
Functional Unit Latency (total/issue)	integer ALU-1/1, load/store-2/1, integer MULT-3/1, integer DIV-12/12, FP adder-2/1, FP MULT-4/1, FP DIV-12/12
D-cache	16k direct-mapped, write-back, write-allocate, 32 byte blocks, 6 cycle miss latency, dual ported, non-blocking interface, 1 outstanding miss per register
Store Buffer	16 elements, non-merging

Table 1: Baseline Simulation Model.

The simulator supports both in-order and out-of-order issue execution models. The in-order issue model provides no renaming and stalls whenever any data hazard occurs on registers. The out-of-order issue model employs a 16 entry register update unit [Soh90] to rename values and hold results of pending instructions. Loads and stores are placed into an 8 entry queue. Stores execute when all operands are ready. Loads may execute when all prior store addresses have been computed; their values come from a matching earlier store in the store queue or from the data cache. Each cycle the register update unit retires 4 results in-order to the architected register file. When stores are retired, the memory value is placed into a store queue and later written to the data cache.

The data cache modeled is a dual ported 16k direct-mapped non-blocking cache. Data cache bandwidth is limited, it can only service two loads or stores each cycle, either speculative or otherwise. Stores are serviced in two cycles using a 16 entry non-merging store buffer. The store buffer retires stored data to the data cache during cycles in which the data cache is unused. If a store executes and the store buffer is full, the entire pipeline is stalled and oldest entry in the store buffer is retired to the data cache.

A number of modifications were made to the simulator to support zero-cycle loads. To compensate for the cost of generating predecode information, I-cache miss latency was increased by two cycles. A BRIC was added with a miss latency of three cycles. During execution, stores employ fast address calculation. Store values are placed speculatively into the store buffer (or store queue) and then committed in the following cycle, after the fast address calculation is verified. Speculating stores improved overall performance by providing store addresses earlier in the pipeline, which reduces the number of memory conflicts with zero-cycle loads. The simulator does not attempt fast address calculation on `register+register` mode loads – it instead performs effective address calculation in the decode stage of the pipeline and accesses the data cache in the execute stage. For all experiments,

Benchmark	Input	Options	Insts (Mil.)	References							
				Loads				Stores			
				Total (Mil.)	Percent of All Loads			Total (Mil.)	Percent of All Stores		
					GP	SP	general		GP	SP	general
Compress	in		62.4	14.9	31.27	6.74	61.97	7.5	23.67	13.27	63.05
Eqntott	int_pri_3.eqn		878.5	211.3	5.24	3.43	91.32	12.9	2.72	63.39	33.87
Espresso	cps.in		491.7	112.9	4.73	3.71	91.55	26.2	0.32	11.51	88.15
GCC	1stmt.i		122.8	28.3	11.47	30.03	58.48	20.2	3.03	42.03	54.93
Sc	loada1		858.4	240.4	17.17	26.79	56.03	103.4	2.44	61.73	35.82
Xlisp	li-input.lsp	(queens 8)	965.2	317.4	18.91	30.44	50.64	177.9	14.24	60.21	25.54
Grep	3x inputs.txt	-E -f regex.in	131.7	42.4	0.67	2.18	97.13	1.5	14.78	44.64	40.56
Perl	tests.pl		193.2	51.1	12.77	37.15	50.07	32.8	9.61	49.01	41.36
YACR-2	input2		386.9	61.8	9.15	31.87	58.97	7.2	0.14	36.53	63.31
Alvinn		NUM_EPOCHS=50	1,238.5	363.7	1.04	1.10	97.84	125.1	0.00	3.09	96.90
Ear	short.m22	args.short	340.4	76.0	1.65	1.01	97.33	44.6	0.16	1.61	98.21
Mdljdp2	mdlj2.dat	MAX_STEPS=150	729.3	278.4	1.49	0.12	98.37	84.9	4.88	0.58	94.52
Mdljsp2	mdlj2.dat	MAX_STEPS=150	875.1	222.1	3.55	0.70	95.73	75.6	9.10	2.37	88.51
Spice2g6	greycode.in	.tran .7n 8n	1,252.0	452.5	32.60	18.61	48.78	76.6	1.85	34.09	64.05
Su2cor	su2cor.in	Short input	824.9	334.4	3.70	2.67	93.61	89.1	0.00	8.00	91.98
Tomcatv		N=129	464.2	172.9	5.64	4.81	89.54	35.8	0.06	0.80	99.13

Table 2: Benchmark programs, inputs, instructions counts, and reference characteristics.

there are only two data cache access ports available each cycle. Data cache ports are arbitrated first to non-speculative loads late in the pipeline, then to the store buffer, and finally, if any are ports are left over, to speculative data cache accesses.

4.1.3 Benchmarks

In selecting benchmarks, we included both integer- and floating point-intensive codes. Table 2 details the programs we analyzed, their inputs, instruction counts, and reference characteristics. The integer codes are in the top group, the floating point codes in the bottom group. Thirteen of the analyzed benchmarks are from the SPEC92 benchmark suite. (We omitted eight of the SPEC92 floating point codes to reduce simulation time.) In addition, we analyzed three other integer codes: *Perl*, a popular scripting language running its test suite, *Grep* performing regular expression matches in a large text file, and *YACR-2*, a VLSI channel router routing a channel with 230 terminals.

4.2 Baseline Performance

In this section, we consider the performance impact of zero-cycle loads on pipelines with one and two decode stages and varied BRIC sizes. As detailed in Section 2, the pipeline implementation with one decode stage (Figure 2) indexes the BRIC in the fetch stage with a load address and has a 2 cycle branch penalty. The pipeline implementation with two decode stages (Figure 4), indexes the BRIC with a register specifier in the first decode stage and has a 3 cycle branch penalty. Figure 6 shows the hit ratios for both address- and register specifier-indexed BRIC of varying size. In each experiment, the BRIC is fully associative with LRU replacement. The figures show only four of the benchmarks, selected as they are representative of the others.

The address-indexed BRIC performs well for even small sizes. The performance is quite good for programs with a large number of static loads, *e.g.*, *GCC* and *Spice*, and programs with a large number of dynamic loads, *e.g.*, *Xlisp* and *Spice*. All the benchmarks hit more than 80% of the time with an 8 entry address-indexed BRIC and more than 90% with a 32 entry address-indexed BRIC. By keeping global and stack pointer loads out of the BRIC, fewer accesses need to use it, and of those remaining, many accesses are within loops which have excellent temporal locality.

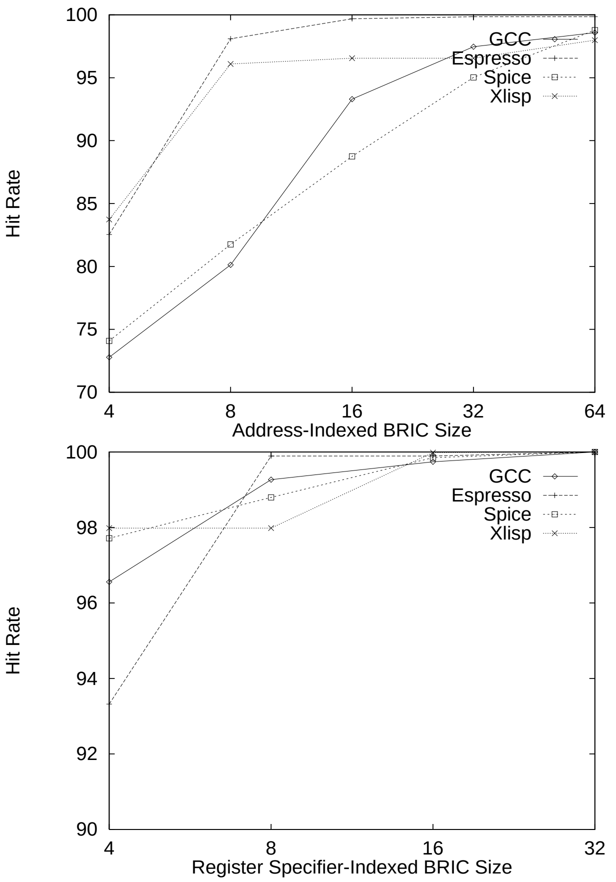

Figure 6: BRIC Hit Rates.

The register specifier-indexed BRIC, used in the two decode stage design, also performs well for all programs. Performance for a even a four entry register specifier-indexed BRIC is very good.

Benchmark	One Decode Stage, Address-Indexed BRIC						Two Decode Stages, Register Specifier-Indexed BRIC			
	Base	Speedup					Base	Speedup		
	IPC	BRIC-8	BRIC-64	No GP/SP	Only GP/SP	Perfect	IPC	BRIC-4	BRIC-8	Perfect
Compress	1.01	1.23	1.23	1.22	1.10	1.40	0.99	1.21	1.22	1.38
Eqntott	1.22	1.41	1.42	1.41	1.03	1.42	1.15	1.37	1.37	1.38
Espresso	1.20	1.43	1.44	1.42	1.03	1.45	1.19	1.40	1.40	1.42
GCC	1.04	1.27	1.28	1.24	1.07	1.41	1.02	1.24	1.25	1.37
SC	1.02	1.48	1.49	1.40	1.16	1.52	1.00	1.45	1.45	1.48
Xlisp	0.99	1.54	1.56	1.48	1.23	1.56	0.98	1.47	1.47	1.48
Grep	0.95	1.95	1.98	1.94	1.01	1.98	0.95	1.96	1.96	1.97
Perl	0.90	1.27	1.28	1.24	1.10	1.33	0.87	1.28	1.28	1.33
YACR-2	1.62	1.37	1.37	1.37	1.12	1.37	1.59	1.36	1.36	1.36
Alvinn	1.02	1.44	1.44	1.44	1.02	1.45	1.01	1.44	1.44	1.45
Ear	0.84	1.29	1.30	1.29	1.01	1.33	0.81	1.29	1.30	1.34
Mdljdp2	0.83	1.30	1.31	1.29	1.01	1.35	0.82	1.29	1.30	1.35
Mdljsp2	0.78	1.17	1.17	1.17	1.01	1.25	0.78	1.18	1.18	1.25
Spice2g6	0.86	1.28	1.29	1.27	1.20	1.65	0.85	1.27	1.27	1.63
Su2cor	0.76	1.13	1.14	1.13	1.01	1.15	0.75	1.12	1.13	1.15
Tomcatv	0.97	1.21	1.22	1.20	1.02	1.27	0.97	1.21	1.21	1.27

Table 3: Zero-Cycle Load Baseline Performance.

This result is to be expected since register file accesses have a significant amount of temporal locality [FP91].

Table 3 shows the results of detailed timing simulations. IPC's of the baseline simulations and speedups are shown for both the one and two decode stage implementation. All experiments were performed with the in-order issue processor model. The speedups shown are the number of cycles for each program to execute with hardware and software support for zero-cycle loads divided by the number of cycles to execute without software and hardware support for zero-cycle loads. For the one decode stage implementation, speedups are shown for both large (64 entry) and small (8 entry) address-indexed BRICs. In each experiment, the BRIC simulated is fully associative and uses LRU replacement.

The speedups are quite impressive, for both the integer and floating point codes. With an 8 entry address-indexed BRIC, we found run-time weighted average speedup of 1.45 for the integer codes and 1.26 for the floating point codes. The speedups for the floating point codes are slightly less than the integer codes because their executions are heavily dominated by other long latencies, *e.g.*, cache miss latencies or floating point computations, which are largely unaffected by zero-cycle load support.

Performance with even a small address-indexed BRIC is quite good. The 64 entry BRIC (column *BRIC-64*) only improves slightly over the 8 entry BRIC simulations (column *BRIC-8*). This result is very positive, suggesting that keeping the BRIC small to reduce processor cycle time impact should not have a significant effect on speedups.

The column labeled *No GP/SP* shows speedups for an 8 entry address-indexed BRIC without separate registers available to cache the global and stack pointer. In this design, the global and stack pointer loads reside in the BRIC as well, reducing its effective capacity, *i.e.*, the design in Figure 2 without the GP and SP registers. The results suggest that special handling of global and stack references only yields marginal improvements in overall performance, more for the programs with a high dynamic frequency of global and stack loads and stores, *e.g.*, *Xlisp* or *Sc*. If designs omit this option, they can still expect good performance.

We also performed experiments to determine the efficacy of the BRIC itself. The column labeled *Only GP/SP* shows the speedups for a configuration without a BRIC, *i.e.*, only global and stack pointer accesses can execute with zero cycle latency. As expected, only the programs which rely heavily on global or stack pointer accesses show any notable speedups.

The right half of Table 3 gives speedups for a pipeline implementation with two decode stages. In this configuration, the branch misprediction penalty is increased to three cycles, and the BRIC is indexed by a register specifier, rather than a load address. Speedups are shown for both a 4 and 8 entry fully associative BRIC with LRU replacement. As suggested by the hit rates in Figure 6, the 8 entry BRIC only offers marginal improvement over the 4 entry BRIC. Overall, the speedups are comparable to the one decode stage design. Either of the presented designs should be equally effective at reducing load latency.

Also shown for both the one and two decode stage designs is program performance with perfect fast address calculation predictions and no BRIC misses (the columns labeled *Perfect*). Perfect performance is only slightly better than actual performance, hence, our current approach is quite effective at exploiting most of the potential performance. Most of the performance loss in each case can be attributed to fast address calculation failures.

4.3 Performance without Software Support

In practice, it may not be possible to employ software support on some or all codes. We examined the performance of zero-cycle loads with and without software support, the results of the experiments are shown in Table 4. All simulations were performed with an in-order issue processor model with two decode stages and an 8 entry fully associative register specifier-indexed BRIC with LRU replacement.

The first column of Table 4, labeled *Impact of S/W*, quantifies the performance impact of our software support on the baseline processor. The numbers shown are the run-time (in cycles) of the programs with software support running on a processor without support for zero-cycle loads over the run-time of the programs without software support running on the same processor. In other words, this result is the impact the user will see running a program with zero-cycle load optimizations on a processor that does not implement zero-cycle loads. The performance impact of our optimizations on the baseline hardware is quite small. We also examined virtual memory performance by simulating a 64 entry TLB

Benchmark	Impact of S/W	Speedup	
		w/SW support	w/o SW support
Compress	1.001	1.22	1.11
Eqntott	1.002	1.37	1.33
Espresso	1.015	1.40	1.39
GCC	1.017	1.25	1.22
SC	1.021	1.45	1.30
Xlisp	1.024	1.47	1.28
Grep	0.991	1.96	1.91
Perl	0.990	1.28	1.19
YACR-2	1.001	1.36	1.29
Alvinn	0.999	1.44	1.43
Ear	0.993	1.30	1.29
Mdljdp2	0.998	1.30	1.27
Mdljsp2	1.000	1.18	1.17
Spice2g6	0.995	1.27	1.08
Su2cor	1.006	1.13	1.12
Tomcatv	1.000	1.21	1.21

Table 4: Performance with and without Software Support.

Benchmark	Speedup			
	In-Order	Out-of-Order	$\frac{Cycle_{In}}{Cycle_{Out}}$	$\frac{Cycle_{In+ZCL}}{Cycle_{Out}}$
Compress	1.22	1.02	1.63	1.34
Eqntott	1.37	1.16	1.26	0.91
Espresso	1.40	1.15	1.65	1.14
GCC	1.25	1.04	1.46	1.16
SC	1.45	1.16	1.69	1.13
Xlisp	1.47	1.10	1.54	1.03
Grep	1.96	1.42	1.67	0.84
Perl	1.28	1.07	1.40	1.08
YACR-2	1.36	1.07	1.70	1.23
Alvinn	1.44	1.09	1.80	1.24
Ear	1.30	1.04	2.10	1.61
Mdljdp2	1.30	1.09	1.85	1.42
Mdljsp2	1.18	1.04	1.55	1.47
Spice2g6	1.27	1.01	1.37	1.36
Su2cor	1.13	1.13	1.68	1.40
Tomcatv	1.21	1.08	1.52	1.41

Table 5: Performance with Out-of-Order Issue.

with random replacement and found that there was no significant increase in the number of TLB misses due to our optimizations.

The third and fourth columns of Table 4 show the speedups attained with support for zero-cycle loads for programs with and without software support. (The third column is reproduced from Table 3.) Program performance, even without software support, is quite good. Programs with many global and stack variables accesses, *e.g.*, *Xlisp* and *SC*, benefit most from software support.

4.4 Performance with Out-of-Order Issue

An aggressive out-of-order issue execution model provides a built-in mechanism for tolerating load latency. Unconstrained by dependencies, the issue mechanism in an out-of-order issue processor is able to run ahead of executing instructions looking for independent operations with which to tolerate latencies. If branch prediction performs well and there is sufficient parallelism, other latency tolerating or reducing techniques should not be required. To determine the effectiveness of this execution model at negating the benefits of zero-cycle loads, we performed experiments comparing the performance of zero-cycle loads running on a processor with and without out-of-order issue capability. The results are shown in Table 5. All simulations were performed on a pipeline with two decode stages using an 8 entry full associative register specifier-indexed BRIC with LRU replacement. All programs are compiled with software support.

As seen by comparing the speedups on an in-order issue processor (in column *In-Order*, reproduced from Table 3) with speedups on an out-of-order issue processor (in column *Out-of-Order*), overall speedups are notably less for the processor using out-of-order issue. This result confirms that the out-of-order issue model is tolerating load latency. However, the resulting speedups are not all insignificant, especially for many of the integer codes. These codes likely have less parallelism available to tolerate load instruction latencies, thus they benefit from the latency reduction offered by fast address calculation.

The rightmost two columns of Table 5 compare the performance of an in-order issue processor with and without zero-cycle load support to an out-of-order issue processor without zero-cycle load support. The column labeled $Cycle_{In}/Cycle_{Out}$ gives the run-time (in cycles) of programs on the in-order issue processor divided by the run-time on the out-of-order issue processor (neither with zero-cycle load support). This metric quantifies the cycle count advantage when running on an out-of-order issue processor. Clearly, the programs take fewer cycles to run on the out-of-order issue processor than on the in-order issue processor. The column labeled $Cycle_{In+ZCL}/Cycle_{Out}$ repeats the experiments, except the in-order issue processor has support for zero-cycle loads. For the integer codes, the performance of the two processors is now much closer – both out performing each other in some cases, with slightly better performance on the out-of-order issue processor.

This result is striking when one considers the clock cycle and design time advantages typically afforded to in-order issue processors. It may be the case that for workloads where untolerated latency is dominated by data cache access latencies (as in the case of the integer benchmarks), an in-order issue design with support for zero-cycle loads may consistently out perform an out-of-order issue processor.

4.5 Performance with Fewer Registers

A number of architectures in wide-spread use today have few architected registers, *e.g.*, the *x86* or *System/370* architectures. To evaluate the efficacy of zero-cycle loads for these architectures, we performed experiments on our extended MIPS architecture, but with only 8 integer and 8 floating point registers (one-quarter the normal supply). The results of the experiments are shown in Table 6. All simulations were performed on a pipeline using in-order issue with two decode stages. All programs are compiled with software support.

The left side of Table 6 shows how a program is affected by reducing the number of architected registers in each register file from 32 to 8. The total number of loads increased by as much as 177%, primarily the result of extra accesses needed to spill and reload temporary variables. The increases are notably larger for programs with larger basic blocks, *e.g.*, the floating point codes, since they typically use more temporary space. Also shown (in the column labeled *Distribution of Extra Loads*) is the breakdown of how much (as a percent of total extra loads) each form of addressing contributes to the overhead.

The column labeled $Cycle_{32-reg}/Cycle_{8-reg}$ quantifies the performance impact of fewer architected registers. It shows the run-time (in cycles) of programs compiled to use 32 registers running on the baseline in-order issue simulator divided by the run-time of the 8 register version of the program running on the same processor. For many of the programs, the performance impact of

Benchmark	Loads				Speedup		
	Percent More Loads	Distribution of Extra Loads: Global	Stack	General	$\frac{Cycle_{32-reg}}{Cycle_{8-reg}}$	$\frac{Cycle_{32-reg}}{Cycle_{8-reg+GPSP}}$	$\frac{Cycle_{8-reg}}{Cycle_{8-reg+ZCL}}$
Compress	23.34	32.41	44.13	23.45	0.85	1.05	1.32
Eqntott	39.82	92.41	7.59	0.00	0.77	0.86	1.40
Espresso	57.11	9.02	47.90	43.08	0.81	0.96	1.45
GCC	24.02	5.74	36.26	58.01	0.90	1.01	1.28
SC	4.67	0.00	72.46	27.54	0.99	1.13	1.43
Xlisp	0.98	100.00	0.00	0.00	0.99	1.20	1.46
Grep	5.46	0.00	48.86	50.96	0.97	1.01	2.03
Perl	26.46	24.26	32.29	43.45	0.86	1.14	1.38
YACR-2	99.06	0.00	63.14	36.86	0.87	1.23	1.67
Alvinn	31.79	0.76	99.24	0.00	0.95	1.18	1.34
Ear	112.36	16.45	34.15	49.40	0.78	0.94	1.50
Mdljdp2	81.41	7.66	45.52	46.82	0.69	0.94	1.36
Mdljsp2	110.21	30.35	69.65	0.00	0.77	0.90	1.33
Spice2g6	9.93	13.96	86.04	0.00	0.95	1.16	1.28
Su2cor	47.20	2.26	97.74	0.00	0.79	0.86	1.13
Tomcatv	177.22	1.62	98.38	0.00	0.52	0.56	1.16

Table 6: Performance with Few Registers.

having more registers is quite large, more so in general for the floating point codes.

Next, we considered 8 register designs with limited and full support for zero-cycle loads. The limited support design only provides cycle zero latency to global and stack pointer accesses. Implementing this support is less costly than full support for zero-cycle loads. (This design is essentially the one in Figure 2 without a BRIC.) This design should perform well considering the predominance of stack and global accesses in the 8 register executions. The speedups in the column labeled $Cycle_{32-reg}/Cycle_{8-reg+GPSP}$ show this implementation's performance with respect to an architecture with 32 registers and no support for zero-cycle loads. For most programs, limited support for zero-cycle loads more than compensates for the lack of registers in the architecture. In most cases, the performance of the register limited architecture is better than its 32 register counterpart. Not only does the zero-cycle load support perform well on the extra accesses due to spills and reloads, but it also performs well on the stack and global pointer accesses that both processors must execute. For a few of the floating point codes, *e.g.*, *Tomcatv*, the improvements rendered still do not approach the performance of the 32 register architecture. These codes suffer from an excessive number of dynamic loads and stores, which saturate available data cache bandwidth and limit overall performance improvements.

The experiments in column $Cycle_{8-reg}/Cycle_{8-reg+ZCL}$ show the performance found with full support for zero-cycle loads on the register-limited architecture. The speedups are shown with respect to the program running on the register-limited architecture without support for zero-cycle loads. As expected, the speedups are better than those found on the 32 register architecture due to the excellent performance of the many extra stack and global accesses.

5 Related Work

The application of early issue as a means of reducing load latency has been gainfully applied in a number of previous works [BC91, EV93, GM93]. The approach used in each of these works is quite similar. An address predictor mechanism, which is a variant of the *load delta table* [EV93], generates addresses early in the pipeline, allowing loads to be initiated earlier than the execute stage. The load delta table tracks both the previous address accessed by a particular load and one or more computed stride values used to predict a load's next effective address. Considering the frequency of strided accesses and accesses with no (zero) stride, the load delta table is an effective approach to predicting load addresses. Our address predictor, fast address calculation, is stateless, eliminating the need for a load delta table. As a result, additional interlocks are not introduced when strides are computed and written into the load delta table. The implementation cost of our approach is also reduced. Our approach features the tightest level of pipeline integration, yielding fewer register and memory interlocks and better potential performance. Tighter pipeline integration, however, does limit the extent to which load latency can be reduced (two cycles in our design). To have an effect on longer load latencies, like those occurring during data cache misses, a load delta table approach may be more effective if used instead or in conjunction with our approach. This observation is further supported by the possibility that the load delta table may perform better on codes where fast address calculation performs poorly (*e.g.*, poorly structured numeric codes).

The C Machine [DM82] used a novel approach to implement zero-cycle access to stack frame variables. At cache misses, memory operand specifiers within instructions were replaced with direct stack cache addresses. When the partially decoded instructions were executed, operands in the stack cache could be accessed as quickly as registers.

In [AVS93], the *knapsack* memory component is presented. Software support was used to place data into the power-of-two aligned knapsack region, providing zero-cycle access to these variables when made with the architecturally-defined knapsack pointer. Zero-cycle access was limited primarily to global data.

Jouppi [Jou89] proposed a pipeline that performed ALU operations and memory access in the same stage. The pipeline employs a separate address generation pipeline stage, pushing the execution of ALU instructions and cache access to the same pipeline stage. This organization increases the mispredicted branch penalty by one cycle. It also removes the load-use hazard that occurs in the traditional 5-stage pipeline, instead introducing an address-use hazard. The address-use hazard stalls the pipeline for one cycle if the computation of the base register value is immediately followed by a dependent load or store. The R8000 (TFP) processor [Hsu94] uses a similar approach. Our approach can be viewed as essentially an extension of this pipeline design. Memory access is pulled back one more stage, to the stage prior to the execute stage. Because we employ fast address calculation, we can do this without increasing the number of address-use hazards.

In [APS95], we presented the design and evaluation of fast address calculation. In this work, we extended the latency reduction capability of our original approach by combining it with a mechanism for early issue. Comparing the baseline results of this paper with those in [APS95], we have roughly doubled the performance improvement for the integer codes and nearly quadrupled the improvement for the floating point codes. These improvements follow from the increased latency reduction of loads from one cycle (with fast address calculation along) to zero cycles (with early issue and fast address calculation) in combination with better overall support for speeding up `register+register` mode loads, an addressing mode which the floating point codes rely on more heavily.

6 Summary and Conclusions

Two pipeline designs supporting zero-cycle loads were presented: an aggressive design for pipelines with a single stage of decode and a less aggressive design for pipelines with multiple decode stages. The designs make judicious use of instruction cache predecode, base register caching, and fast address calculation to produce load results two cycles earlier than traditional pipeline designs. The design with multiple decode stages was markedly simpler because more of the component operations of a load could be performed after fetching the load instruction.

We evaluate these designs in a number of contexts: with and without software support, in-order vs. out-of-order issue, and on architectures with many and few registers.

Overall, we found the speedups afforded by zero-cycle loads, for either pipeline design, were excellent. For the one decode stage design with in-order issue, we found a cycle-weighted speedup of 1.45 for the integer codes and 1.26 for the floating point codes. Speedups were good for even small BRIC sizes. Software support was generally effective, more so on the integer codes, and even without software support, speedups were still quite good.

Speedups on the out-of-order issue processor were significantly less due to the latency tolerating capability of the execution model. However, some programs still showed notable speedups, likely because the executions lacked sufficient parallelism to tolerate all load latency, and thus benefited from the latency reduction capability of fast address calculation. An in-order issue processor with zero-cycle load support compared favorably in performance to an out-of-order issue processor for programs with significant untolerated load instruction latency.

With fewer registers, the frequency of loads and their impact of program performance increases significantly, especially for floating point codes. Providing an 8 register architecture with zero-cycle load support for only global and stack pointer references, resulted in performance comparable to a 32 register architecture. This result suggests limited support for zero-cycle loads is one avenue available to improving the performance of legacy architectures with few registers. With full support for zero-cycle loads, speedups were quite good, slightly better than for the 32 register architecture due to excellent prediction performance for the many extra global and stack accesses present.

We feel the consistent and impressive performance advantage of zero-cycle loads, especially for improving the performance of in-order issue processors or architectures with few registers, makes this approach an attractive choice for future processor designs.

Acknowledgements

We thank Dionisios Pnevmatikatos, Scott Breach, Alain Kägi, and the anonymous referees for their comments on earlier drafts of this paper. This work was supported in part by NSF Grants CCR-9303030 and MIP-9505853, ONR Grant N00014-93-1-0465, and a donation from Intel Corp.

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[APS95] T. M. Austin, D. N. Pnevmatikatos, and G. S. Sohi. Streamlining data cache access with fast address calculation. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[AVS93] T. M. Austin, T.N. Vijaykumar, and G. S. Sohi. Knapsack: A zero-cycle memory hierarchy component. Technical Report TR 1189, Computer Sciences Department, UW–Madison, Madison, WI, November 1993.

[BC91] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. *Supercomputing '91*, pages 176–186, 1991.

[CCH+87] F. Chow, S. Correll, M. Himelstein, E. Killian, and L. Weber. How many addressing modes are enough. *Conference Proceedings of the Second International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 117–121, October 1987.

[DM82] D. R. Ditzel and H. R. McLellan. Register allocation for free: the C machine stack cache. In *1st International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 48–56, Palo Alto, CA, March 1982.

[EV93] R. J. Eickemeyer and S. Vassiliadis. A load-instruction unit for pipelined processors. *IBM J. Res. Develop.*, 37(4):547–564, July 1993.

[FP91] M. Farrens and A. Park. Dynamic base register caching: A technique for reducing address bus width. *Proceedings of the 18th Annual International Symposium on Computer Architecture*, 19(3):128–137, May 1991.

[GM93] M. Golden and T. Mudge. Hardware support for hiding cache latency. CSE-TR-152-93, University of Michigan, Dept. of Elect. Eng. and Comp. Sci., February 1993.

[Gwe94a] L. Gwennap. Digital leads the pack with 21164. *Microprocessor Report*, 8(12):1–10, September 1994.

[Gwe94b] L. Gwennap. MIPS R10000 uses decoupled architecture. *Microprocessor Report*, 8(14):18–22, October 1994.

[Hsu94] P. Y.–T. Hsu. Designing the TFP microprocessor. *IEEE Micro*, 14(2):23–33, April 1994.

[Jou89] N. P. Jouppi. Architecture and organizational tradeoffs in the design of the MultiTitan CPU. *Proceedings of the 16st Annual International Symposium on Computer Architecture*, 17(3):281–289, May 1989.

[KH92] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[Soh90] G. S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Trans. on Computers*, 39(3):349–359, March 1990.

[WJ94] S. J.E. Wilton and N. P. Jouppi. An enhanced access and cycle time model for on-chip caches. Tech report 93/5, DEC Western Research Lab, 1994.

[WRP92] T. Wada, S. Rajan, and S. A. Pyzybylski. An analytical access time model for on-chip cache memories. *IEEE Journal of Solid-State Circuits*, 27(8):1147–1156, August 1992.