Streamlining Data Cache Access with Fast Address Calculation

Todd M. Austin   Dionisios N. Pnevmatikatos   Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706
{austin,pnevmati,sohi}@cs.wisc.edu

Abstract

For many programs, especially integer codes, untolerated load instruction latencies account for a significant portion of total execution time. In this paper, we present the design and evaluation of a fast address generation mechanism capable of eliminating the delays caused by effective address calculation for many loads and stores.

Our approach works by predicting early in the pipeline (part of) the effective address of a memory access and using this predicted address to speculatively access the data cache. If the prediction is correct, the cache access is overlapped with non-speculative effective address calculation. Otherwise, the cache is accessed again in the following cycle, this time using the correct effective address. The impact on the cache access critical path is minimal; the prediction circuitry adds only a single OR operation before cache access can commence. In addition, verification of the predicted effective address is completely decoupled from the cache access critical path.

Analyses of program reference behavior and subsequent performance analysis of this approach shows that this design is a good one, servicing enough accesses early enough to result in speedups for all the programs we tested. Our approach also responds well to software support, which can significantly reduce the number of mispredicted effective addresses, in many cases providing better program speedups and reducing cache bandwidth requirements.

1 Introduction

Successful high-performance processor implementations require a high instruction completion rate. To achieve this goal, pipeline hazards must be minimized, allowing instructions to flow uninterrupted in the pipeline. Data hazards, an impediment to performance caused by instructions stalling for results from executing instructions, can be minimized by reducing or tolerating functional unit latencies. In this paper, we focus on a pipeline optimization that reduces the latency of load instructions, resulting in fewer data hazards and better program performance.

There are many contributing factors to the latency of a load instruction. If a load hits in the data cache, the latency of the operation on many modern microprocessor architectures is 2 cycles: one cycle to compute the effective address of the load, and one cycle to access the data cache. If the load does not hit in the data cache, the latency is further increased by delays incurred with accessing lower levels of the data memory hierarchy, *e.g.*, cache misses or page faults.

Figure 1 illustrates how load instruction latency can affect program performance. The figure shows a traditional 5-stage pipeline executing three dependent instructions. The pipelined execution continues without interruption until the `sub` instruction attempts to use the result of the previous `load` instruction. In a traditional 5-stage pipeline, a load instruction requires the EX stage for effective address calculation and the MEM stage for cache access. The result of the load operation is not available until the end of cycle 5 (assuming a single cycle cache access and the access hits in the data cache). This situation creates a data hazard on register `rw`, forcing the dependent `sub` instruction to stall one cycle. As a result, the EX stage of the pipeline is idle in cycle 5 – if the latency of the load instruction were only one cycle, the code sequence would complete one cycle earlier.

Much has been done to reduce the performance impact of load latencies. The approaches can be broadly bisected into two camps: techniques which assist programs in tolerating load latencies, and techniques which reduce load latencies. Tolerating load latencies involves moving independent instructions into unused pipeline delay slots. This reallocation of processor resources can be done either at compile-time, via instruction scheduling, or at run-time with some form of dynamic processor scheduling, such as decoupled, dataflow, or multi-threaded. For a given data memory hierarchy, a good approach to reducing load latencies is through better register allocation. Once placed into a register, load instructions are no long required to access data.

There are, however, limits to the extent to which existing approaches can reduce the impact of load latencies. Tolerating techniques require independent work, which is finite and usually quite small in the basic blocks of control intensive codes, *e.g.*, many integer codes [AS92]. The current trend to increase processor issue widths further amplifies load latencies because exploitation of instruction level parallelism decreases the amount of work between load instructions. In addition, tolerating these latencies becomes more difficult since more independent instructions are required to fill pipeline delay slots. Global scheduling techniques [Fis81, MLC$^+$92, ME92] have been developed as a way to mitigate this effect. Latency reduction in the form of register allocation is limited by the size and addressability of register files, forcing many program variables into memory.

Our approach, called *fast address calculation*, works to reduce load instruction latency by allowing effective address calculation to proceed in parallel with cache access, thereby eliminating the extra cycle required for address generation. The technique uses a simple circuit to quickly predict the portion of the effective address needed to speculatively access the data cache. If the address is predicted correctly, the cache access completes without an extra cycle for address

Instruction	Clock Cycle 1	2	3	4	5	6	7
`add  rx,ry,rz`	IF	ID	EX	WB			
`load rw,4(rx)`		IF	ID	EX	MEM	WB	
`sub  ra,rb,rw`			IF	ID	stall	EX	WB

Figure 1: Example of an Untolerated Load Latency.

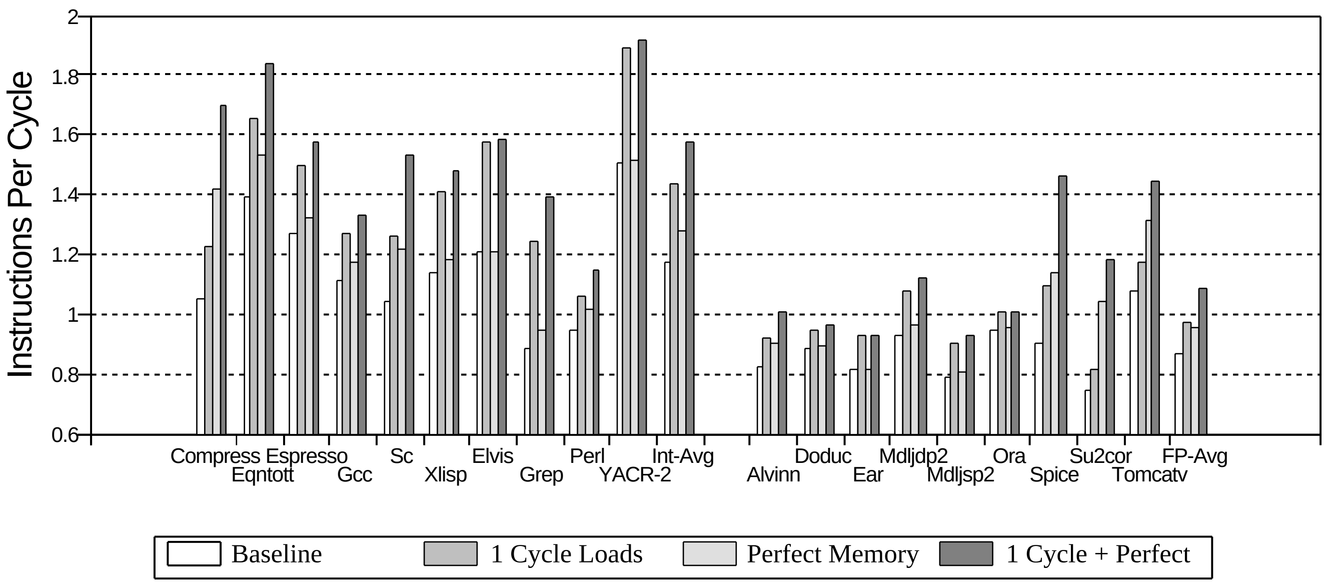

Figure 2: Impact of Load Latency on IPC.

calculation. If the address is mispredicted, the cache is accessed again using the correct effective address. The predictor is designed to minimize its impact on cache access time, adding only a single OR operation before cache access can commence. Verification of the predicted effective address is completely decoupled from the cache access critical path. Our approach benefits from software support: a compiler and linker conscious of our fast address calculation mechanism is shown to significantly reduce the number of mispredicted addresses.

To gauge the performance potential of fast address calculation, we compared the IPC of 19 benchmarks executing with varied load latencies. The experiments were performed on an aggressive 4-way in-order issue superscalar processor simulator with a 16k direct-mapped non-blocking data cache, with 32 byte blocks, and a 6 cycle miss delay. The programs were compiled with GNU GCC for an extended MIPS target. GCC employs local instruction scheduling and aggressive priority-based register allocation, thus these numbers are representative of an environment where the compiler works to tolerate and eliminate load latencies. (Refer to Section 5 for a detailed description of our experimental framework.) The results of our experiments are shown in Figure 2. *Baseline* shows program performance with 2-cycle loads and a 6 cycle cache miss, *1-Cycle Loads* reduces the cache hit latency to 1 cycle but retains the 6 cycle cache miss penalty, *Perfect Cache* represents a 2 cycle load latency and a 0 cycle cache miss penalty, and *1 Cycle+Perfect* represents the case where all load instructions complete in 1 cycle. In addition, the graph shows the average IPC, weighted by program run-time (in cycles), for the integer codes (the left group) and the floating point codes.

The extra cycle used for effective address calculation (as seen by comparing *1-Cycle Loads* to *Baseline*) has a sizable impact on the performance of all of the programs tested. Generally, the integer codes saw more improvement from 1 cycle loads than the floating point codes. This result is to be expected considering the better cache performance and shorter average functional unit latencies of the integer codes. Moreover, the relative performance impact of 1 cycle loads is greater than executing with 2 cycle loads and a perfect cache for more than half of the programs.

Clearly, the extra cycle used for effective address calculation is a performance bottleneck, in many cases a larger one than that of cache misses. In the remainder of this paper, we develop our fast address calculation approach and examine its impact on program performance. In Section 2, we present analyses of program reference behavior, the primary motivating factor behind the design of our fast address generation mechanism. Section 3 describes our approach to predicting effective addresses and the pipeline modifications required. Section 4 describes ways in which software can increase the prediction accuracy of the fast address generation mechanism. Results of analyses of 19 benchmarks are presented in Section 5. Finally, Section 6 describes related work and Section 7 presents a summary and conclusions.

2 Program Reference Behavior

The case for our fast address generation approach can be made by examining the reference behavior of programs. We profiled the load instructions of several benchmarks compiled for an extended MIPS architecture. (The benchmarks and architecture are detailed in Section 5.) We made a number of key observations which are detailed below.

Benchmark	Insts (Mil.)	Total Refs (Millions)		Loads		
		Loads	Stores	% Global	% Stack	% General
Compress	61.5	14.3	7.5	29.23	9.21	61.56
Eqntott	875.7	205.2	12.6	5.08	7.09	87.83
Espresso	474.4	109.1	25.9	3.91	5.26	90.83
Gcc	121.7	25.8	19.7	7.35	36.02	56.63
Sc	840.1	217.3	91.8	12.68	33.97	53.36
Xlisp	965.2	290.0	172.2	16.78	42.33	40.90
Elvis	249.3	67.7	28.6	1.63	6.33	92.04
Grep	122.2	42.1	1.5	1.13	3.64	95.23
Perl	203.6	50.0	34.2	10.69	43.15	46.16
YACR-2	386.8	59.0	7.1	7.61	32.72	59.68
Alvinn	1015.4	362.5	125.1	0.73	1.51	97.77
Doduc	1597.2	536.3	195.8	29.33	38.44	32.23
Ear	338.4	75.6	43.0	1.04	1.19	97.76
Mdljdp2	729.1	276.9	84.9	2.30	0.23	97.47
Mdljsp2	874.4	219.8	75.6	5.01	1.14	93.86
Ora	1057.1	231.2	98.2	33.19	33.14	33.67
Spice	1250.6	443.9	76.5	27.42	21.03	51.55
Su2cor	796.1	333.8	88.8	2.91	3.76	93.32
Tomcatv	464.2	172.8	35.9	4.68	4.07	91.25

Table 1: Program Reference Behavior.

2.1 Reference Type

There are three prevalent modes of addressing that occur during execution, which we classify as *global pointer*, *stack pointer*, and *general pointer* addressing. Table 1 details the dynamic number of loads and stores executed by each program and the dynamic breakdown by reference type for loads.

Global pointer addressing is used to access small global (static) variables. The MIPS approach to global pointer addressing uses a reserved immutable register, called the *global pointer*, plus a constant offset to access variables in the *global region* of the program's data segment [CCH$^+$87]. The linker constructs the global region such that all variables referenced by name are grouped together near the target address of the global pointer. As shown in Table 1, global pointer addressing is prevalent in some programs, but not all. The frequency of this mode is highly dependent on the program structure and style.

Stack pointer addressing is used to access elements of a function's stack frame. The stack pointer register holds an address to the base of the stack frame of the currently executing function. Accesses to frame elements are made using the stack pointer register plus a constant offset (positive, by convention, on the MIPS architecture). As is the case with global pointer addressing, stack pointer addressing is also a prevalent, but not an entirely dominating form of addressing.

The third mode of addressing, general pointer addressing, encompasses all other accesses. These accesses are the result of pointer and array dereferencing occurring during program execution. Quantitatively, all the benchmarks make heavy use of general pointer addressing with more than half of them using it for more than 80% of loads.

2.2 Offset Distribution

A RISC-style load has two inputs: base and offset. The base is added to the offset during effective address computation. In our extended MIPS architecture, the base is supplied by a register and the offset may be supplied by either a signed 16-bit immediate constant, *i.e.*, `register+constant` addressing, or via a register, *i.e.*, `register+register` addressing.

We examined the size distribution of offsets for global, stack, and general pointer accesses. The cumulative size distributions (using a log scale) are shown in Figure 3 for four of the benchmarks. (These

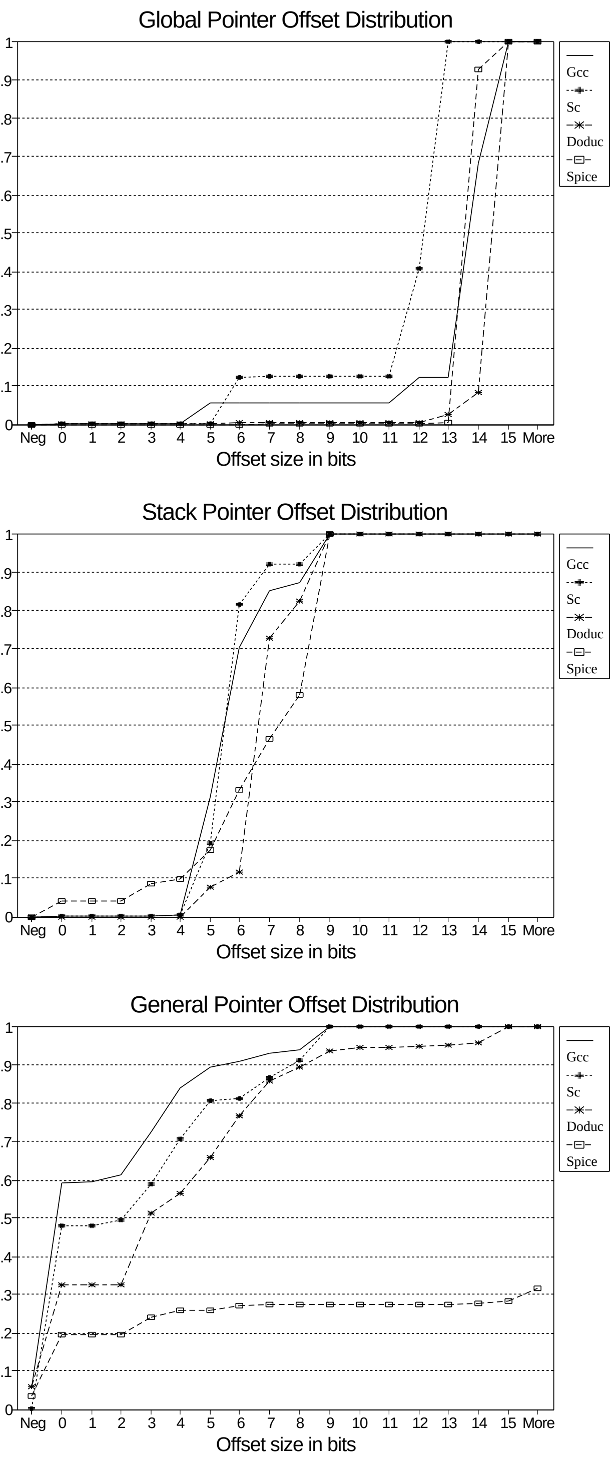

Figure 3: Load Offset Cumulative Distributions.

curves are representative of the other benchmarks.) The graphs include loads using `register+register` addressing, in which case the base and offset of the load are determined by convention.

The offsets applied to the global pointer are typically quite large, being that they are partial addresses. As one would expect, there is a strong correlation between the size of the offsets required and the aggregate size of the global data addressed by the program.

Stack pointer offsets tend to be large as well due to the large size of stack frames. While a stack frame may have only a few local variables, there are overheads not apparent to high-level language programmers which can greatly increase its size. These overheads include register save areas, dynamic stack allocations, return address storage, among others.

For general pointer accesses, most load offsets are small. In fact, for a number of programs we analyzed, *e.g.*, *Alvinn* and *Espresso*, zero was the most common offset used. Zero offsets are primarily the product of array subscript operations where strength reduction [ASU86] of the subscript expression succeeded, pointer dereferences to basic types (*e.g.*, integers), and pointer dereferences to the first element of a structured (record) variable.

Non-zero offsets arise from primarily three sources: structure offsets, some array accesses, and array index constants. Structure offsets are small constants applied to pointers when accessing fields of a structured variable. Array base addresses are combined with index values to implement array accesses, *e.g.*, `a[i]`. Our compiler, based on GNU GCC, only generates this form of addressing when strength reduction of the subscript expression is not possible or fails. (When strength reduction is successful, a zero offset suffices.) Index constants are generated when part of an array subscript expression is constant, *e.g.*, `array[i+10]`. In addition, the compiler creates many index constants when unrolling loops. Index constants are usually small, although when in the higher dimension of a multi-dimensional array, they can become large.

For a few of the floating point programs, most notable *Spice* and *Tomcatv*, there were a significant number of large offsets. This result indicates strength reduction of array accesses was generally ineffective. Consequently, the compiler had to rely on the brute force approach of adding the index variable to the base address of the array for every array access made, creating many large (index register) offsets.

Negative offsets are usually small immediate constants, generated by negative array subscript constants. They occur infrequently for both the integer and floating point intensive programs, *e.g.* for GCC they account for 5.7% of the general pointer loads and about 3.2% of all loads.

To summarize these observations, it is clear that any mechanism designed to speed up address calculation must: 1) perform well on all reference types, 2) perform well on small offsets, and 3) perform well on large offsets applied to the stack and global pointers. Secondary goals to good performance include support for predicting large index register offsets and support for small negative offsets. In the following section, we present our fast address calculation mechanism, designed to satisfy these criteria while minimizing cost and impact to processor designs.

3 Fast Address Calculation

The fast address calculation mechanism predicts effective addresses early in the pipeline, thereby allowing loads to complete earlier. To accomplish this task, we exploit an organizational property of on-chip data caches.

To minimize access time, on-chip caches are organized as wide two-dimensional arrays of memory cells (as shown in Figure 4). Each row of the 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 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. Hence, on-chip cache organizations require the set index portion of the effective address early in the clock cycle and the block offset and tag portion late – after the cache row and tag have been read. Our prediction mechanism leverages off this property of on-chip caches, allowing part of the address calculation to proceed in parallel with cache access.

Figure 4 shows a straightforward implementation our effective address prediction mechanism for an on-chip direct-mapped cache, targeting ease of understanding rather than optimal speed or integration. 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.[1] Because many offsets are small, 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 are used to compute the block offset and tag portion of the effective address. Later in the cache access cycle, the block offset is used by the multiplexor to select the correct word from the cache row, and the tag portion of the effective address is compared to the tag value read from the tag array.

Special care is taken to accommodate small negative offsets. The set index portion of negative offsets must be inverted, otherwise address prediction will fail. In addition, the prediction will fail if a borrow is generated into the set index portion of the effective address computation. In our design, we've assumed that offsets from the register file arrive too late for set index inversion, thus address predictions for these loads and stores fail if the offset is negative. This conservative design decision has little impact on our results since negative index register offsets are extremely infrequent.

To complete the hardware design, we augment the cache hit/miss detection logic with a circuit that verifies the predicted address. Using the result of this circuit, the cache controller and the instruction dispatch mechanism can determine if the access needs to be re-executed in the following cycle using the non-speculative effective address (computed in parallel with the speculative cache access). A misprediction is detected by looking for carries, either propagated into or generated in the set index part of the effective address computation. Four failure condition exist: 1) a carry (or borrow) is propagated out of the block offset portion of the effective address (signal *Overflow* in Figure 4), 2) a carry is generated in the set index portion of the effective address (signal *GenCarry*), 3) a constant offset is negative and too large (in absolute value) to result in an effective address within the same cache block as the base register address (signal *LargeNegConst*), or 4) an offset from the register file is negative (signal *IndexReg<31>*).

Figure 5 illustrates our approach to fast effective address generation. Example (a) shows a pointer dereference. Since the offset is zero, no carry is generated during address calculation and the predicted address is correct. Example (b) shows an access to a global variable though the global pointer. In this example, the global pointer is aligned to a large power of two, so carry-free addition is sufficient to generate the correct address. In example (c), carry-free addition is sufficient to predict the portion of the address above the block offset, but full addition is required to compute the block offset. Since a carry in not generated out of the block offset portion of the effective address computation, the prediction succeeds. Finally, example (d) shows a stack frame access with a larger offset. In this case, the predicted address is incorrect because a carry is propagated out of the block offset and generated in the set index portion of the effective address computation.

[1] Technically, a carry-free addition requires an XOR function, but use of a simpler inclusive OR suffices here because the functions only differ when address prediction fails.

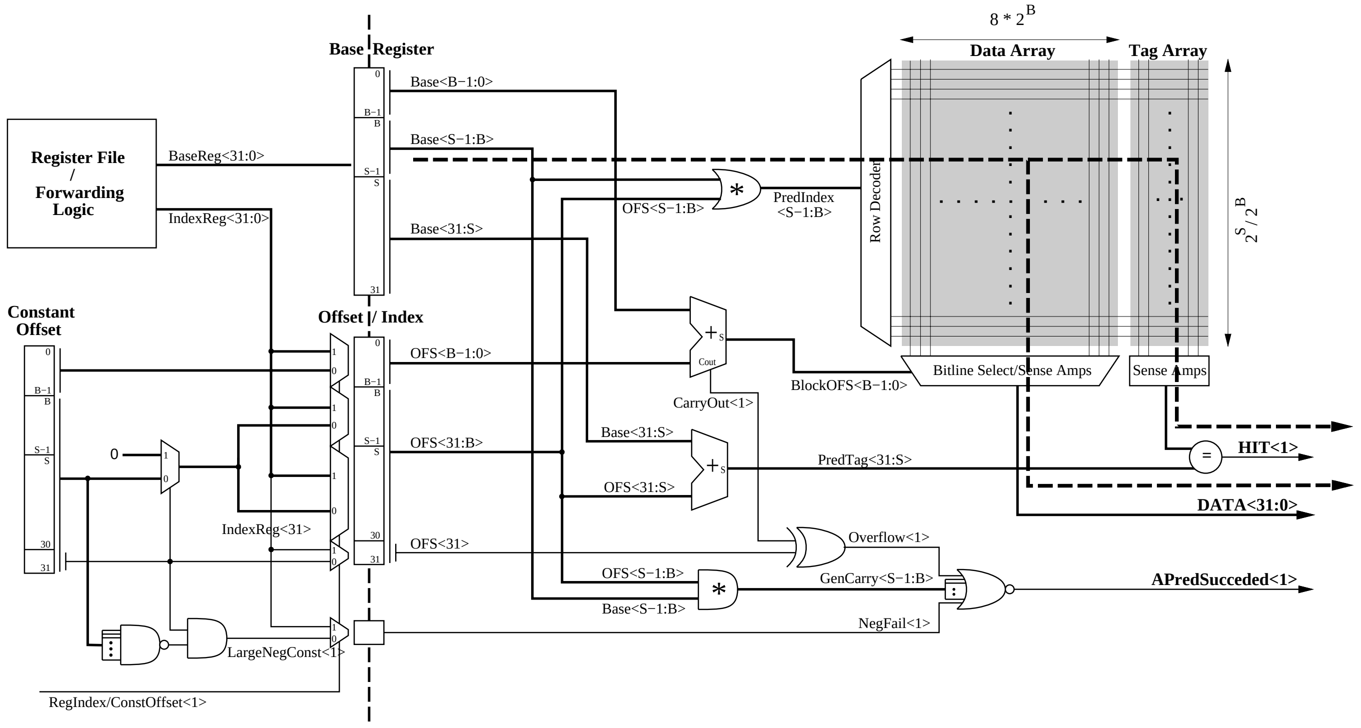

Figure 4: Pipeline Support for Fast Address Calculation. Bold lines indicate a bus, gates with an '*' signify a replicated gate for every line of the connected bus. 2^S is the size of a cache set in bytes, 2^B is the block size. The architecture shown has 32-bit addresses.

```
                                     <-- Tag -->  <-- Index-->  <BO>
     load r3, 0(r8)
     r8            = 0x0001ac00    000...01 10   10 1100 0000  0000
(a)  offset        = 0x00000000    000...00 00   00 0000 0000  0000
     prediction    = 0x0001ac00    000...01 10   10 1100 0000  0000
     actual result = 0x0001ac00    000...01 10   10 1100 0000  0000

     load r3, 2436(gp)
     gp            = 0x00010000    000...01 00   00 0000 0000  0000
(b)  offset        = 0x00000984    000...00 00   00 1001 1000  0100
     prediction    = 0x00010984    000...01 00   00 1001 1000  0100
     actual result = 0x00010984    000...01 00   00 1001 1000  0100

     load r3, 102(sp)
     sp            = 0x7fff5b84    011...11 01   01 1011 1000  0100
(c)  offset        = 0x00000066    000...00 00   00 0000 0110  0110
     prediction    = 0x7fff5bea    011...11 01   01 1011 1110  1010
     actual result = 0x7fff5bea    011...11 01   01 1011 1110  1010
                                                      Generated
     load r3, 364(sp)                                 Carry
     sp            = 0x7fff5b84    011...11 01   01 1011 1000  0100
(d)  offset        = 0x0000016c    000...00 00   00 0001 0110  1100
     prediction    = 0x7fff5be0    011...11 01   01 1011 1110  0000
     actual result = 0x7fff5cf0    011...11 01   01 1100 1111  0000

                                           Generated   Propagated
                                           Carry       Carry
```

Figure 5: Examples of Fast Address Calculation. The address bits are split into the tag, index and block offset fields corresponding to a 16K byte direct-mapped data cache with 16 byte blocks.

3.1 Design Considerations

We've designed our prediction mechanism to minimize impact on the cache access critical path. The two typical cache access critical paths are denoted with bold dashed lines in Figure 4. While a much more detailed design would be required to demonstrate the exact impact our prediction mechanism has on the cache access critical path, we will point out a few design features of this circuit that indicate it should have minimal impact on cycle time. Three paths through our prediction circuit could affect the cache access cycle time. The first is through the tag adder. The tag portion of the effective address computation must arrive at the tag comparator before the tag array output becomes stable. For small addresses and large cache set sizes, this computation will likely complete before tag comparison. For large addresses and small cache set sizes, this computation may not complete in time. For these designs, the logical OR operation used to compute the set index could also be used to compute the tag portion of the address. We have run all our experiments with and without full addition capability in the tag portion of the effective address computation and found this capability to be of limited value. This result is to be expected considering the relatively small size of load offsets compared to cache set sizes, and the large alignments required on either the base or offset for carry-free addition to succeed on the set index portion of the address computation but fail on the tag portion.

The second path that could affect cycle time runs through the block offset adder. This result must arrive at the data array multiplexor before the data array produces stable output. For most cache designs, a 4- or 5-bit adder should suffice for this computation. The complexity of the block offset adder is small, on the order of the cache row decoders, hence, this part of our design will likely not impact the cache access critical path.

The third path that could affect cache access cycle time is through the prediction verification circuit. Since this circuit is completely decoupled from the normal cache access, it cannot affect the cache access cycle time as long as validation of the predicted effective address completes before the end of the clock cycle. This prediction verification signal, however, could affect processor cycle time if it is not available early enough to allow the processor control logic to schedule operations for the following cycle (a function of the success and failure of memory accesses in the current cycle). Since the verification circuit is very simple, we do not expect it to impact the processor cycle time.

The OR operation used to generate the upper portion of the effective address is, however, directly on the critical paths denoted in Figure 4. The impact of this operation may or may not impact processor cycle time, depending on the specifics of the pipeline design.

In some designs, it may be possible to integrate the OR operation into the address decoder logic or the execute stage input latches, possibly reducing cycle time impact to zero.

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, our approach will not work.

Another important consideration is the handling of stores. The question of whether or not to speculate stores really depends on specifics of the memory system. If stalling a store can cause a load to stall, *e.g.*, the processor executes all memory accesses in order, performance may be improved by speculatively executing stores. For designs that speculatively execute stores, care must be taken to ensure that a misspeculated store can be undone. For designs employing a store buffer [Jou93], this may not pose a problem, as the approach uses a two-cycle store sequence. In the first cycle the tags are probed to see if the access hits in the cache, in the second (possibly much later cycle) the store is made to the cache. Since our fast address calculation mechanism determines the correctness of the address after one cycle, the store buffer entry can simply be reclaimed or invalidated if the effective address is incorrect. Possibly more important to consider, however, is the effect of increased bandwidth demand due to misspeculated accesses on store buffer performance. With speculative cache accesses stealing away free cache cycles, the processor may end up stalling more often on the store buffer, possibly resulting in overall worse performance. We examine one detailed design that addresses a number of these issues in Section 5.

4 Increasing Prediction Performance with Software Support

Software support can increase the prediction accuracy of our fast address generation mechanism by reducing the need for full-strength addition in the set index portion of the effective address calculation. We accomplish this task by decreasing the size of offset constants and increasing the alignment of base pointers. Note, however, software support is only used as a mechanism to improve the performance of our approach, it is not required. As we show in Section 5, our fast address calculation mechanism is remarkably resilient, providing good speedups even without software support.

Compiler support was added to GNU GCC (version 2.6.0), linker support was added to GNU GLD (version 2.3). The modifications made for each type of addressing is slightly different, ensuring a high prediction rate for each.

Global Pointer Accesses

Since the linker controls the value of the global pointer and offsets applied to it, it is trivial to ensure all global pointer accesses are correctly predicted. The linker limits all offsets off the global pointer to be positive and relocates the global region to an address starting at a power-of-two boundary larger than the largest offset applied. As a result, carry-free addition suffices for any global pointer access.

Stack Pointer Accesses

As is the case with global pointer addressing, the compiler completely controls the value of the stack pointer and the organization of stack frames. By aligning the stack pointer and organizing the stack frame so as to minimize the size of offset constants, it is possible to ensure that all stack pointer accesses are correctly predicted.

The compiler maintains alignment of the stack pointer by restricting all frame sizes to be a multiple of a program-wide stack pointer alignment. At function invocations, the adjusted frame size is subtracted from the current stack pointer. Since the stack pointer is initially aligned in the startup code to the program-wide alignment, the alignment is maintained throughout the entire execution.

Nearly all stack pointer addressing is performed on scalar variables. By sorting the elements of the stack frame such that the scalars are located closest to the stack pointer, the compiler can minimize the size of offsets constants applied to the stack pointer.

Using this approach, carry-free addition suffices for address computations in which the offset is smaller than the alignment of the stack pointer. Some programs, most notably the FORTRAN codes, have a large variance in frame sizes, and thus benefit little from a program-wide stack pointer alignment. For stack frames larger than the program-wide stack pointer alignment, the compiler employs an alternative approach: the stack pointer is explicitly aligned to a larger alignment by AND'ing the stack pointer with the adjusted power-of-two frame size times a negative one. Since this approach creates variable size stack frames, a frame pointer is required for access to incoming arguments not in registers. In addition, the previous stack pointer value must be saved at function invocation and restored when the function returns.

The impact of this approach is increased stack memory usage – frame size overhead can grow as much as 50%. If a program uses more memory, cache and virtual memory performance could suffer. We provide the programmer a compiler option to limit the size of alignments enforced on the stack pointer, thereby providing a means to control memory overhead.

General Pointer Accesses

For general pointer accesses, offsets are typically small and positive, the result of index constants and structure offsets. The compiler increases the likelihood of a carry not being generated out of the block offset portion of the effective address by aligning variable allocations to a multiple of the cache block size.

Global and local variable alignments are increased to the next power-of-two larger than the size of the variable, bounded by the block size of the target cache. Dynamic storage alignments are increased in the same manner by the dynamic storage allocator, *e.g.*, `malloc()`. Since many languages, *e.g.*, C, employ type-less dynamic storage allocation, the allocator lacks the type information required to minimize alignment overheads. As a result, all dynamic allocations are aligned to the maximum allowed alignment, typically the block size of the target cache. `Alloca()` allocations (used heavily by the benchmarks *GCC* and *Grep*) employ a similar approach for dynamic storage allocation within stack frames.

To ensure proper alignments of interior objects, *e.g.*, array elements, the compiler rounds up the size of structured types to the next larger power of two, bounded by the block size of the target cache. Since basic types, *e.g.*, integers and floats, are already a power of two in size, overheads are only incurred for arrays of structured variables. The compiler does not, however, enforce stricter alignments on structure fields, as this would serve to spread out elements of a structure. Our experiments indicated that having dense structures is a consistently bigger win than enforcing stricter alignments within structured variables.

As is the case with larger stack frame alignments, these techniques can increase memory usage by as much as 50%. Hence, a compiler option was provided to limit the alignments placed on variable addresses and sizes.

In addition to the changes described above, we also made modifications to GCC's existing optimization routines to improve the performance of optimized code. Specifically, we modified common subexpression elimination (CSE) to give preference to aligned pointer subexpressions. Small modification were also made to the strength-reduction phase of loop optimization. We modified the address cost functions so as to make `register+register` addressing seem very expensive. This modification makes GCC work harder to strength-reduce loop induction variables, resulting in more zero offset loads and stores within loops.

5 Experimental Evaluation

We evaluated the effectiveness of our approach by examining its ability to predict the effective addresses generated by 19 non-trivial programs. To examine the efficacy of software support, we performed the experiments on programs compiled with and without fast address calculation specific optimizations. In addition, we investigated the impact of fast address calculation in the context of a realistic processor model by examining each program's performance running on a detailed superscalar timing simulator extended to support fast address calculation.

5.1 Experimental Framework

All experiments were performed with programs compiled for an extended MIPS architecture. The architecture is functionally identical to the MIPS-I ISA [KH92], except for the following differences:

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

The addition of `register+register` addressing allowed us to more fairly evaluate the benefits of fast address calculation. We found that without this mode some programs, *e.g.*, *spice2g6*, showed dramatic performance improvements – the result of which was not due to the merits of fast address calculation, but rather to the program's need for `register+register` addressing. This addressing mode is not supported in the base MIPS ISA, but can be efficiently synthesized using fast address calculation.[2] We removed all architected delay slots to simplify the implementation of a detailed superscalar timing simulator.

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). We added fast address calculation specific optimizations to GCC; these additions include compiler arguments to control: structure element alignment, structure size alignment, stack frame size alignment, static allocation alignment, and dynamic allocation alignment. In addition, we extended GLD to enforce global pointer alignments. The compiler and linker support required was surprisingly simple, totaling less than 1000 lines of code. The Fortran codes were first converted to C using AT&T F2C version 1994.09.27; we were careful to configure F2C such that it promoted all local scalar variables from C statics to true local variables (automatics).

When performing fast address calculation specific optimizations, we optimized both the library codes as well as the program code. The following fast address calculation specific optimizations were applied:

- global pointer alignment: GLD aligned the global pointer to a power-of-two value larger than the largest relocation applied to it. All global pointer relocations were restricted to be positive. (Normally, the initial value of the global pointer is dependent on the size of the data segment and is not aligned.)
- stack pointer alignment: GCC rounded all stack frame sizes up to the next multiple of 64 bytes, resulting in a program-wide stack pointer alignment of 64 bytes. (Normally, GCC maintains an 8 byte alignment on the stack pointer.) Frames larger than 64 bytes enforce larger stack pointer alignments of up to 256 bytes by explicitly aligning the stack pointer on function invocation and restoring the original value on function return.

[2]The MIPS-I ISA, with 2-cycle loads, can synthesize `register+register` addressing using an `add` and a zero-offset `lw` in 3 cycles. The same architecture, with support for fast address calculation, can synthesize the mode in 2 cycles.

Benchmark	Input	Options/Modifications
Compress	in	
Eqntott	int_pri_3.eqn	
Espresso	cps.in	
GCC	1stmt.i	
Sc	loada1	
Xlisp	li-input.lsp	Short input (queens 8)
Elvis	unix.c	%s/for/forever/g, %s/./& /g
Grep	3x inputs.txt	-E -f regex.in
Perl	tests.pl	
YACR	input2	
Alvinn		NUM_EPOCHS=50
Doduc	doducin	
Ear	short.m22	args.short
Mdljdp2	mdlj2.dat	MAX_STEPS=150
Mdljsp2	mdlj2.dat	MAX_STEPS=250
Ora		ITER=60800
Spice2g6	greycode.in	.tran .7n 8n
Su2cor	su2cor.in	Short input
Tomcatv		N=129

Table 2: Benchmark programs and their inputs.

- static variable alignment: static allocations were placed with an alignment equal to the next power-of-two larger or equal to the size of the variable, not exceeding 32 bytes.
- dynamic variable alignments: `malloc()` and `alloca()` allocation alignments were increased from the default of 8 to 32 bytes.
- structured variable internal alignments: internal structure offsets were not changed, however, structure sizes were increased to the next power-of-two larger than or equal to the normal structure size, with the overhead not exceeding 16 bytes.

5.2 Analyzed Programs

In selecting benchmarks, we tried to maintain a good mix between integer and floating point codes. Table 2 details the programs we analyzed and their inputs. Fifteen of the analyzed benchmarks are from the SPEC92 benchmark suite [SPE91]. In addition, we analyzed four other integer codes: *Elvis*, a VI-compatible text editor performing textual replacements in batch mode, *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.

5.3 Prediction Performance

The left side of Table 3 shows the baseline statistics for our benchmark programs without software support. Shown are the number of instructions, execution time in cycles on our baseline simulator (*i.e.*, a 4-way superscalar without fast address generation support – described in Section 5.5), total loads and stores executed, instruction and data cache miss ratios for 16k byte direct-mapped caches with 32 byte blocks, and total memory size.

The right side of Table 3 lists the prediction failure rates for all loads and stores for 16 and 32 byte cache block sizes, *i.e.*, the case where the prediction circuitry is able to perform 4 or 5 bits of full addition in the block offset portion of the effective address computation, respectively.

Overall, the percentage of incorrect predictions is quite high, reflecting the case that many pointers are insufficiently aligned to allow for carry-free addition in the set index part of the effective address

Benchmark	Insts	Cycles	Loads	Stores	Miss Ratio		Mem	Failed Predictions (percent)			
					I-cache	D-cache	Usage	Block Size 16		Block Size 32	
	(Mil.)	(Mil.)	(Mil.)	(Mil.)	32b	32b		Load	Store	Load	Store
Compress	61.5	58.4	14.3	7.5	0.00	15.65	438k	66.24	52.28	66.24	51.36
Eqntott	875.7	627.4	205.2	12.6	0.00	4.70	2704k	8.63	52.84	8.40	50.06
Espresso	474.4	374.0	109.1	25.9	0.16	2.54	400k	12.25	8.14	11.06	7.75
Gcc	121.7	109.5	25.8	19.7	1.63	3.09	1416k	49.94	42.87	47.10	38.88
Sc	840.1	811.3	217.3	91.8	0.17	7.11	493k	40.50	58.98	32.08	55.68
Xlisp	965.2	850.6	290.0	172.2	0.71	1.72	115k	50.21	54.30	43.44	50.83
Elvis	249.3	207.1	67.7	28.6	0.50	0.44	90k	5.90	9.96	4.77	7.48
Grep	122.2	139.0	42.1	1.5	0.03	3.88	377k	4.53	45.63	2.93	44.99
Perl	203.5	214.4	50.0	34.2	3.63	4.63	3625k	45.04	43.40	42.50	41.94
YACR-2	386.9	261.0	59.0	7.1	0.01	0.67	195k	13.91	44.85	13.24	43.07
Alvinn	1015.4	1236.2	362.5	125.1	0.02	4.21	507k	2.44	3.07	2.30	2.86
Doduc	1597.2	1820.5	536.3	195.8	1.55	2.26	144k	73.01	68.68	71.62	66.33
Ear	338.4	416.5	75.6	43.0	0.00	0.02	208k	20.58	25.18	11.18	12.79
Mdljdp2	729.1	787.3	276.9	84.9	0.00	1.52	267k	31.41	16.93	27.35	16.92
Mdljsp2	874.4	1110.7	219.8	75.6	0.00	1.52	227k	29.32	10.43	28.98	10.43
Ora	1057.1	1112.9	231.2	98.2	0.00	0.33	50k	69.52	75.29	65.10	69.91
Spice	1250.6	1388.9	443.9	76.5	0.36	10.16	3227k	86.72	35.71	86.45	35.54
Su2cor	796.1	1073.3	333.8	88.8	0.08	23.55	4131k	26.10	40.24	24.74	36.04
Tomcatv	464.2	431.6	172.8	35.9	0.01	8.63	945k	44.21	32.19	43.60	31.09

Table 3: Program statistics without software support.

calculation. Some of the programs, however, have very low prediction failure rates, *e.g.*, *Elvis* and *Alvinn*. For these programs, the frequency of zero-offset loads is very high, indicating that prediction is working fairly well because effective address computation is not required. Overall the prediction failure rate decreased when the block size increased, since misaligned pointers benefited from more full addition capability in our fast address calculation mechanism.

5.4 Impact of Software Support

Now we turn our attention to the potential of the software – compiler and linker – to improve the prediction accuracy of our fast address calculation mechanism. The detailed program statistics for the benchmarks compiled with fast address calculation specific optimizations are shown in Table 4. The left hand side of the table shows the percent change in instruction count, cycle count (on the baseline simulator without fast address calculation), number of loads and stores, and total memory size with respect to the program without fast address calculation optimizations (the results in the left hand side of Table 3). For the instruction and data cache (16k byte direct-mapped), the table lists the absolute change in the miss ratio.

Generally, fast address calculation specific optimizations did not adversely affect program performance on the baseline simulator. The total instruction count as well as the number of loads and stores executed are roughly comparable. The cycle count differences (without fast address calculation support) were small; the largest difference was 1.76% more cycles for *GCC*. Cache miss ratios saw little impact for both the instruction and data caches.

We also examined total memory usage, as it is an indirect metric of virtual memory performance. The largest increases experienced were for *Perl*, *Espresso*, and *Xlisp* where memory demand increased by as much as 20%. However, the absolute change in memory consumption for these programs was reasonably small, much less than a megabyte for each. In addition, we examined TLB performance running with a 64 entry fully associative randomly replaced data TLB with 4k pages and found the largest absolute difference in the miss ratio to be less than 0.1% (for *Perl*). Given these two metrics, we do not expect software support to adversely impact the virtual memory performance of these programs.

As the right hand side of Table 4 shows, our software support was extremely successful at decreasing the failure rate of effective address predictions. Comparing the prediction failure rates in Table 4 (labeled "All") to the 32 byte block failure rates of Table 3, we see that the percentage of loads and stores mispredicted is consistently lower, the prediction failure rate decreasing by more than 50% in some cases.

A number of the programs, *e.g. Spice* and *Tomcatv*, still possessed notably high address misprediction rates. To better understand their cause, we profiled loads and stores to determine which were failing most. The two dominating factors leading to address prediction failures were:

- array index failures: Many loads and stores using `register+register` addressing resulted in failed predictions. Our compiler only uses this addressing mode for array accesses, and then only when strength-reduction fails or is not possible, *e.g.*, an array access not in a loop. (If strength-reduction is successful, `register+constant` addressing suffices.) As one would expect, array index values are typically larger than the 32 byte alignment placed on arrays, resulting in high prediction failure rates. Table 4 shows (under "No R+R") the prediction failure rate for all loads and stores except those using `register+register` mode addressing. For many programs, array index operations are clearly a major source of mispredicted addresses.

- domain-specific storage allocators: a number of programs, most notably *GCC*, used their own storage allocation mechanisms, this led to many pointers with poor alignment and increased prediction failure rates.

These factors, however, are not without recourse. We are currently investigating a number approaches aimed at limiting their effect. A strategy for placement of large alignments should eliminate many

Benchmark	Insts % Change	Cycles % Change	Loads % Change	Stores % Change	Miss Ratio Change		Mem Usage %Change	Failed Predictions (percent)			
								Loads		Stores	
					I-cache	D-cache		All	No R+R	All	No R+R
Compress	-0.38	+0.50	+0.00	-0.01	-0.00	+0.00	+1.14	23.54	0.00	8.86	3.18
Eqntott	+0.40	+0.15	+1.39	-0.04	-0.00	+0.00	+0.11	1.33	1.25	15.40	15.40
Espresso	-0.33	-0.59	-0.01	-0.03	-0.00	-0.00	+13.25	3.57	2.52	1.08	1.14
Gcc	+0.86	+1.76	+1.01	+0.90	-0.02	+0.00	+1.20	15.71	15.51	9.16	9.00
Sc	-0.06	+0.12	-0.01	-0.02	-0.00	+0.00	+1.01	3.53	3.45	16.28	16.27
Xlisp	-0.09	+0.80	-0.12	-0.08	-0.01	+0.00	+15.65	1.12	1.12	1.30	1.32
Elvis	-0.17	+0.27	-0.30	-0.36	-0.01	-0.00	+1.11	1.36	0.99	2.42	2.42
Grep	-0.72	-1.14	-0.95	-24.82	-0.00	-0.00	+3.18	1.08	0.72	3.41	3.51
Perl	-0.95	+0.22	-0.53	+0.71	-0.04	+0.01	+20.03	13.31	13.31	11.33	11.09
YACR-2	+0.42	-0.17	+2.33	-0.01	-0.00	-0.00	+0.00	3.66	3.29	23.24	23.67
Alvinn	+0.12	-0.03	-0.01	-0.00	-0.00	-0.00	+0.20	0.92	0.92	1.77	1.77
Doduc	+0.13	+0.21	+0.06	-0.25	-0.02	-0.00	+2.78	20.92	14.11	30.13	28.49
Ear	+0.04	+0.09	+0.18	-0.02	-0.00	+0.00	+2.40	12.19	12.19	12.14	12.14
Mdljdp2	+0.02	-0.34	+0.05	-0.00	-0.00	-0.00	+1.87	25.50	5.45	0.18	0.01
Mdljsp2	-0.07	+0.03	+0.10	+0.01	-0.00	+0.00	+0.44	22.17	0.44	0.29	0.01
Ora	+0.24	+1.51	+0.00	+0.00	-0.00	+0.00	+10.00	20.28	16.96	1.18	1.18
Spice	-0.13	+0.46	+0.12	+0.01	-0.00	+0.00	+0.28	38.00	5.15	12.11	8.26
Su2cor	+0.59	+0.54	+0.24	+0.51	-0.00	-0.00	+0.12	22.16	6.62	30.22	18.86
Tomcatv	+0.00	+0.03	-0.00	+0.00	-0.00	+0.00	+0.32	38.15	0.04	40.13	35.35

Table 4: Program statistics with software support. The cache block size is 32 bytes.

array index failures.[3] In addition, program tuning could rectify many mispredictions due to domain-specific allocators.

5.5 A Simulation Case Study

Prediction performance does not translate directly into program runtime improvements. A successful effective address prediction may or may not improve program performance, depending on whether or not the access is on the program's critical path. For example, if a correct prediction creates a result a cycle earlier but the result is not used until many cycles later, program performance will be unaffected. A true measure of performance impact requires a much more detailed analysis. One possible measure is effective load latency, however, this metric still does not reflect the processor's ability to tolerate (part of) memory latency. To gauge the performance of our fast address calculation approach in the context of a realistic processor model, we analyzed the performance of the benchmarks running on a detailed superscalar timing simulator extended to support fast address calculation.

Our baseline simulator (used to generate the cycle counts shown in Table 3 and 4) is detailed in Table 5. The simulator executes all user-level instructions; it implements a detailed timing model of a 4-way in-order issue superscalar microprocessor and the first level of instruction and data cache memory. The pipeline model we implemented is a traditional 5 stage model, *i.e.*, all ALU operations begin execution in the third stage of the pipeline (EX) and non-speculative loads and stores execute in the fourth stage (MEM), resulting in a non-speculative load latency of 2 cycles. The data cache modeled is a dual ported 16k direct-mapped non-blocking cache. Data cache bandwidth is not unlimited, it can only service up to two loads or one store 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.

Fetch Width	4 instructions
Fetch Interface	able to fetch any 4 contiguous instructions per cycle
I-cache	16k direct-mapped, 32 byte blocks, 6 cycle miss latency
Branch Predictor	1024 entry direct-mapped BTB with 2-bit saturating counters, 2 cycle misprediction penalty
Issue Mechanism	in-order issue of up to 4 operations per cycle, allows out-of-order completion, can issue up to 2 loads or 1 store per cycle
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, two read ports, one write port (dual ported via replication), non-blocking interface, 1 outstanding miss per register
Store Buffer	16 elements, non-merging

Table 5: Baseline Simulation Model.

A number of modifications were made to the simulator to support fast address calculation. Stores are allowed to execute speculatively, in fact, we want them to as our processor model requires that all loads and stores execute in order. Delaying a store to the MEM stage of the pipeline could have the effect of delaying a later load. When a store is executed, it is entered into a store buffer queue.

[3]For example, in the case of *Spice* aligning a single large array to its size would eliminate nearly all mispredictions.

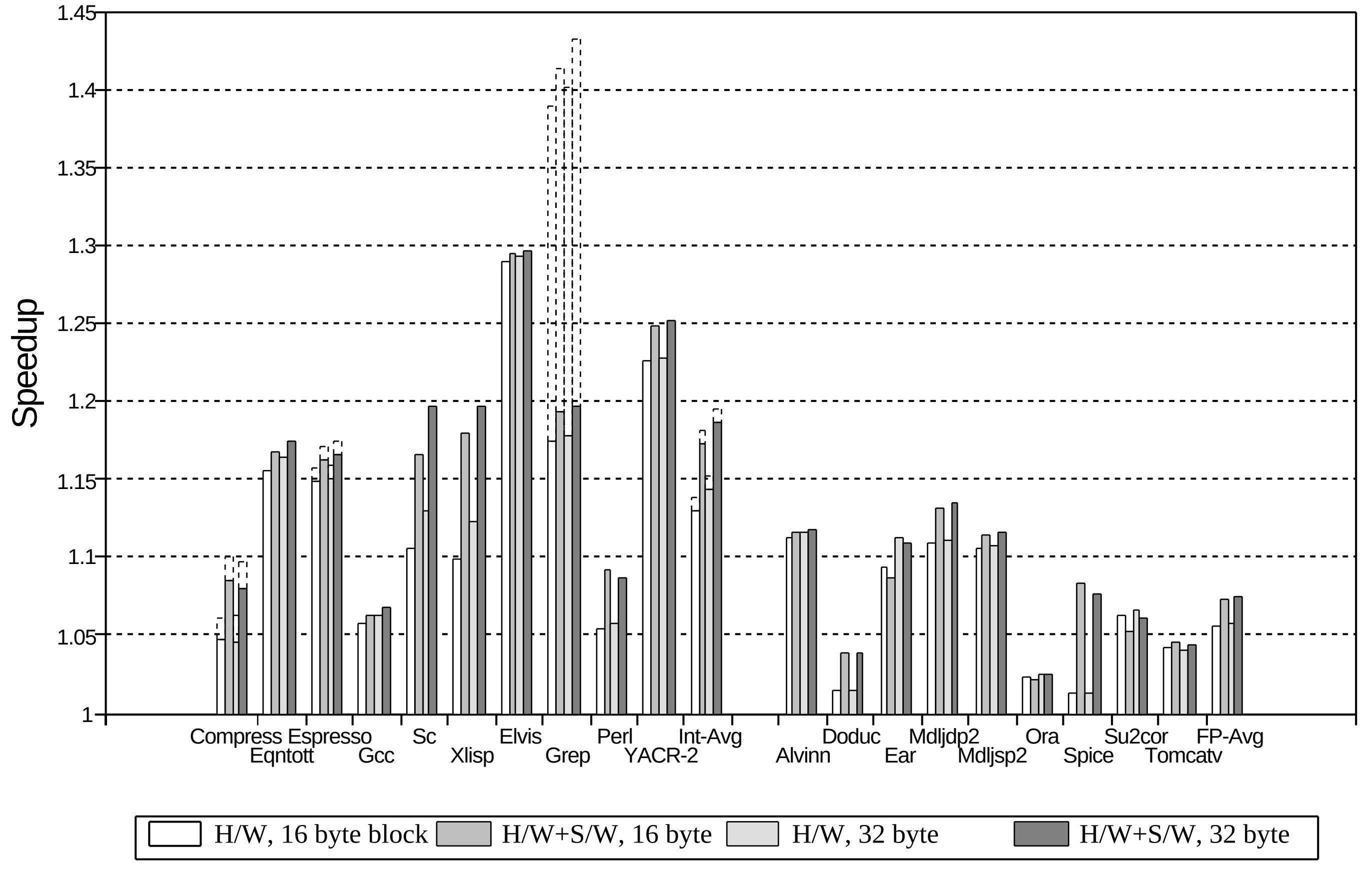

Figure 6: Speedups, with and without software support over baseline model execution time for a 16K byte data cache with 16 and 32 byte blocks. The dashed bars indicate where there was improvement with `register+register` mode speculation.

If the store address is mispredicted, the store is re-executed in the following cycle and its address in the store buffer is updated.

When a memory access is mispredicted in the EX stage of the modified pipeline, the access must re-execute in the MEM stage. In this event, we continue to issue instructions into the pipeline; however, loads and stores issued in the following cycle stall their data cache access until the MEM stage. Thus, even if the load or store would have been correctly speculated, there is no benefit. The only exception to this rule is that a load may speculatively execute immediately after a misspeculated load. The advantage of this issue strategy is that work continues to progress into the pipeline whenever possible. In addition, program performance will not degrade over the baseline model if address prediction fails often (assuming sufficient data cache bandwidth).

Figure 6 shows execution speedups as a function of three design parameters: with and without software support, with 16 and 32 byte blocks, and with and without `register+register` mode speculation. Also shown are the average speedups for the integer and floating point codes, weighted by the run-time (in cycles) of the program. All speedups are computed with respect to the execution time (in cycles) of the baseline program (no fast address calculation specific optimizations) running on the baseline simulator.

On the average, fast address calculation without software support improves the performance of integer programs by 14%, largely independent of block size or `register+register` mode speculation. The floating point programs show a smaller speedup of 6%. This is a very positive result – even without software support, one could expect program performance to consistently improve, and by a sizable margin for integer codes.

The combination of software and hardware manages to give somewhat better performance improvements. We found an average speedup of 19% for all integer programs, with no individual program speedup less than 6%. For the floating point programs, speedups were smaller in magnitude with an average of 7.5%. The compiler optimizations have a positive effect on most programs, and tend to assist more where the hardware-only approach is ineffective, (*e.g.*, Compress).

The consistent speedup across all programs is a very important property of fast address calculation for it allows the designer a trade-off between a longer cycle time and increased performance for integer programs. For example, if fast address calculation increases the cycle time by 5%, the average floating point performance will still improve slightly while the average integer performance will improve by a sizable 13.5%.

We performed all simulations with 16 and 32 byte cache blocks, *i.e.* where the prediction circuitry is able to perform 4 or 5 bits of full addition in parallel with cache access, respectively. The impact of increasing the block size was positive but small in magnitude for most programs, resulting in an overall difference of less than 3% for all experiments. In all cases, the improvement in the average performance was less than 1%.

Considering the prediction failure rate of `register+register` mode addressing, we examined program performance with and without speculation of this mode. The only programs which experienced any change in performance were *Compress*, *Espresso*, and *Grep*. *Grep*'s stellar performance improvement is the result of many `register+register` accesses to small arrays which benefit from limited full addition in the block offset portion of address computa-

Benchmark	R+R Speculation		No R+R Speculation	
	Hardware Only	Software Support	Hardware Only	Software Support
Compress	24.21	16.47	8.92	+0.00
Eqntott	3.02	1.40	2.95	1.33
Espresso	5.38	3.06	3.95	2.01
Gcc	20.85	7.26	20.31	6.65
Sc	13.10	2.71	13.10	2.65
Xlisp	17.43	1.17	17.43	1.17
Elvis	2.49	1.31	2.46	1.04
Grep	2.63	1.10	1.94	0.40
Perl	19.40	7.98	18.55	7.41
YACR-2	5.07	3.87	4.68	3.49
Alvinn	1.33	1.00	1.33	1.00
Doduc	22.68	13.49	17.37	7.28
Ear	8.95	10.32	8.95	10.32
Mdljdp2	21.11	19.56	7.06	3.28
Mdljsp2	18.81	16.32	2.68	+0.00
Ora	24.72	14.03	21.03	10.97
Spice	45.86	32.44	7.46	3.07
Su2cor	19.92	20.33	7.65	5.72
Tomcatv	32.52	33.56	4.22	2.77

Table 6: Memory Bandwidth Overhead, numbers shown are the total failed speculative cache accesses as a percentage of total references.

tion. Average speedup increased less than 1% for the integer codes and was unchanged for the floating point codes. The overall lackluster improvement is the result of high failure rates when predicting `register+register` mode addresses. Without a means to effectively predict `register+register` mode loads and stores, their speculation appears to have little overall benefit, especially in light of increased demand on cache bandwidth.

Table 6 shows the increase in the number of accesses to the data cache (in percent of total accesses). These numbers reflect memory accesses that were mispredicted and actually made during execution; in other words, these results are the overhead in cache accesses due to speculation. Without compiler support, a large fraction of the speculative memory accesses are incorrect (as shown in Table 3), requiring more cache bandwidth, as much as 45% for *Spice*. The compiler optimizations cut down this extra bandwidth significantly. For most programs the increase in the required cache bandwidth is less than 10%, and the maximum increase in bandwidth is less than 34%; without `register+register` mode speculation the bandwidth increases are at most 11%. Despite the increase in cache accesses due to speculation, the impact of store buffer stalls was surprisingly small, typically less than a 1% degradation in the speedups attained with unlimited cache store bandwidth. (The results in Figure 6 include the performance impact of store buffer stalls.)

A fitting conclusion to our evaluation is a comparison between the performance of our implementation and the performance potentials explored in Figure 2. Realized performance with respect to the potential performance of 1 cycle loads is quite good. With software support, most programs realized at least half of the performance potential, with more than half of the programs exceeding 80%. Comparing the speedup of the integer programs (19% with hardware and software support) to the speedup with a perfect cache (only 8%) reveals a perhaps more striking result – fast address calculation consistently outperforms a perfect cache with 2 cycle loads.

6 Related Work

Golden and Mudge [GM93] explored the use of a load target buffer (LTB) as a means of reducing load latencies. An LTB, loosely based on a branch target buffer, uses the address of a load instruction to predict the effective address early in the pipeline. They conclude the cost of the LTB is only justified when the latency to the first level data cache is at least 5 cycles. Our approach has two distinct advantages over the LTB. First, our approach is much cheaper to implement, requiring only a small adder circuit and a few gates for control logic. Second, our approach is more accurate at predicting effective addresses because we predict using the operands of the effective address calculation, rather than the address of the load. In addition, we employ compile-time optimization to further improve performance.

An earlier paper by Steven [Ste88] goes as far as proposing a 4-stage pipeline that eliminates the address generation stage and executes both memory accesses and ALU instructions in the same stage. Steven proposes the use of an OR function for all effective address computation. Steven's approach was only intended as a method for speeding up stack accesses, all other accesses require additional instructions to explicitly compute effective addresses. The performance of this pipeline organization was not evaluated.

Jouppi [Jou89] considers a notably different pipeline organization that has a separate address generation pipeline stage, and pushes the execution of ALU instructions to the same stage as cache access. This pipeline organization removes the load-use hazard that occurs in the traditional 5-stage pipeline, but instead introduces an address-use hazard and increases the mispredicted branch penalty by one cycle. 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] adopts this pipeline in an attempt to assist the instruction scheduler in tolerating load delay latencies. Hsu argues that the nature of floating point code (the target workload of the R8000) provides more parallelism between address calculation and loads than between loads and the use of their results. In a recent paper, Golden and Mudge [GM94] compare this pipeline organization (which they name "AGI") with the traditional 5-stage pipeline (which they name "LUI") to determine which is more capable of tolerating load latency. They found that for a single issue processor with short load latencies and dynamic branch prediction, an AGI pipeline performs slightly better than a LUI pipeline, as long as the branch prediction accuracy is more than 80%. In spite of this observation, both pipelines still suffer from many untolerated load latencies.

AMD's K5 processor [Sla94] overlaps a portion of effective address computation with cache access. The lower 11 bits of the effective address is computed in the cycle prior to cache access. The entire 32 bit effective address is not ready until late into the cache access cycle, just in time for the address tag check.

The idea of exploiting the two-dimensional structure of memory is being used in several other contexts, such as paged mode DRAM access [HP90]. A strong parallel to this work can be found in [KT91]. Katevenis and Tzartzanis propose a technique for reducing pipeline branch penalties by rearranging instructions so that both possible targets of a conditional branch are stored in a single I-cache line. The high bandwidth of the I-cache is used to fetch both targets of a branch instruction. The branch condition is evaluated while the I-cache is accessed and the condition outcome is used to late-select the correct target instruction.

7 Concluding Remarks

In this paper we presented the design and evaluation of fast address calculation, a novel approach to reducing the latency of load instructions. The approach works by predicting early in the pipeline the effective address of a memory access and using this predicted address to speculatively access the data cache. If the prediction is correct, the cache access is overlapped with non-speculative effective address calculation. Otherwise, the cache is accessed again in the following cycle, this time using the correct effective address.

The predictor's impact on the cache access critical path is minimal. The prediction circuitry adds only a single OR operation before cache access can commence. In addition, verification of the predicted effective address is completely decoupled from the cache access critical path.

Our evaluation shows that without software support, the prediction accuracy of the basic hardware mechanism varies widely. For the 19 programs examined, prediction success rates ranged from 13 to 98%. However, detailed timing simulations of the programs executing on a superscalar microprocessor resulted in consistent program speedups – an average speedup of 14% for the integer codes and 6% for the floating point codes. With the addition of simple compiler and linker support, prediction accuracy increased significantly, with success rates ranging from 62 to 99%. Simulated performance with software support increased as well, resulting in average speedup of 19% for the integer codes and 7.5% for the floating point codes. We also measured the increase in cache bandwidth due to access speculation and found it was generally very low for our design. With software support, speculation required at most 34% more accesses. By preventing `register+register` addressing mode speculation, cache bandwidth requirements drop to at most 11% more accesses, with little impact on overall performance.

We feel the consistent performance advantage of fast address calculation coupled with the low cost of its use, in terms of hardware support, software support, and cache bandwidth demand, makes this approach an attractive choice for future processor designs.

Acknowledgements

We thank Scott Breach, Jim Goodman, Alain Kägi, Jim Larus, Andreas Moshovos, T.N. Vijaykumar, David Wood, and the anonymous referees for their comments on earlier drafts of this paper. This work was supported by grants from the National Science Foundation (grant CCR-9303030) and Office of Naval Research (grant N00014-93-1-0465).

References

[AS92] Todd M. Austin and Gurindar S. Sohi. Dynamic dependency analysis of ordinary programs. In *Conference Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 342–351. Association for Computing Machinery, May 1992.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[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.

[Fis81] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transaction on Computers*, C-30(7):478–490, July 1981.

[GM93] Michael Golden and Trevor Mudge. Hardware support for hiding cache latency. Cse-tr-152-93, University of Michigan, Dept. of Electrical Engineering and Computer, February 1993.

[GM94] Michael Golden and Trevor Mudge. A comparison of two pipeline organizations. *Proceedings of the 27th Annual International Symposium on Microarchitecture*, 25(1):153–161, November 1994.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Mateo, CA, 1990.

[Hsu94] Peter Yan–Tek Hsu. Designing the TFP microprocessor. *IEEE Micro*, 14(2):23–33, April 1994.

[Jou89] Norman 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.

[Jou93] Norman P. Jouppi. Cache write policies and performance. *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 21(2):191–201, May 1993.

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

[KT91] Manolis Katevenis and Nestoras Tzartzanis. Reducing the branch penalty by rearranging instructions in a double-width memory. *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 19(2):15–27, April 1991.

[ME92] Soo-Mook Moon and Kemal Ebcioğlu. An efficient resource-constrained global scheduling technique for superscalar and VLIW processors. *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 55–71, December 1992.

[MLC$^+$92] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Conference Record of the 25th Annual International Symposium on Microarchitecture*, pages 45–54, Portland, OR, December 1992. Association for Computing Machinery.

[Sla94] Michael Slater. AMD's K5 designed to outrun Pentium. *Microprocessor Report*, 8(14):1–11, October 1994.

[SPE91] SPEC newsletter, December 1991.

[Ste88] Gordon Steven. A novel effective address calculation mechanism for RISC microprocessors. *Computer Architecture News*, 16(4):150–156, September 1988.

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

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