Reprinted from *Proceedings of the Hawaii International Conference on System Sciences*, January 1993

Memory Bandwidth Optimizations for Wide-Bus Machines

Michael A. Alexander, Mark W. Bailey, Bruce R. Childers, Jack W. Davidson, and Sanjay Jinturkar

Department of Computer Science, University of Virginia

Abstract

One of the critical problems facing designers of high-performance processors is the disparity between processor speed and memory speed. This has occurred because innovation and technological improvements in processor design have outpaced advances in memory design. While not a panacea, some gains in memory performance can be had by simply increasing the width of the bus from the processor to memory. Indeed, high-performance microprocessors with wide buses (i.e., capable of transferring 64 bits or more between the CPU and memory) are beginning to become available (e.g., MIPS R4000, DEC Alpha, and Motorola 88110). This paper discusses some compiler optimizations that take advantage of the increased bandwidth available from a wide bus. We have found that simple strategies can reduce the number of memory requests by 10 to 15 percent. For some data and compute intensive algorithms, more aggressive optimizations can yield significantly higher reductions.

1: Introduction

A serious problem facing computer architects is the mismatch between processor speed and memory bandwidth. Processor speed has grown much faster than the speed of memory devices, consequently most high-performance processors are starved for memory bandwidth. For example, processors operating at speeds of 66 to 100 megahertz are available now and implementations that operate in the 100 to 200 megahertz range have been announced. On the other hand, current DRAM cycle times are an order of magnitude slower than processor cycle times [HEN90] . Providing a cache constructed of faster SRAM parts reduces the performance gap, but caches cannot eliminate the problem. Furthermore, for processors used in scientific computing applications, the large problem sizes can make caches less effective.

Consequently, organizations and technologies that reduce the CPU-DRAM performance gap are of high, current interest. A simple way to increase memory bandwidth is to double the size of the bus. Indeed, over the years microprocessor bus size has grown from 8-bits, to 16-bits, and to 32-bits. Microprocessors with external/internal buses that are 64-bits wide are just beginning to appear (e.g. MIPS R4000, Motorola 88110, and DEC Alpha). These machines have the ability to fetch 64-bit quantities in a single memory-unit operation. While increasing the bus width doubles the available memory bandwidth, most programs will only use a fraction of the available bandwidth. For example, we expect that most programmers will continue to use 32-bit integers and only use 64-bit integers when more precision is needed. Certainly for existing code, it seems likely that integers will remain 32 bits in length. Thus, to take full advantage of a wide bus will require new code improvement techniques. This paper describes and evaluates the effectiveness of some code improvement techniques that are designed to take advantage of wide-bus machines (WBMs). That is, a microprocessor with a memory bus width at least twice the size of the integer data type handled by the processor and assumed by the programmer[†].

Our investigations show that with no changes to the compiler technology, WBMs can expect reductions in memory bus cycles on the order of five to fifteen percent. Using new code improvement algorithms designed to exploit the availability of a wide bus, our studies show, that for many memory-intensive algorithms, it is possible to reduce the number of memory loads and stores by 30 to 40 percent.

1.1: Related Work

Because the disparity between processor speed and memory speed has become a performance bottleneck, eliminating this bottleneck is the focus of many researchers. The approaches to solving the problem can be categorized as either software or hardware solutions. This paper focuses solely on software solutions. The interested reader is

†Using this definition, a microprocessor with 16-bit integers (e.g., PDP-11 and Intel 80286) and a 32-bit bus would be a wide-bus architecture.

referred to Patterson and Hennessy [HEN90] and Goodman [GOO83] for a discussion of system-oriented hardware approaches to solving the problem. Bursky [BUR92] , Nakagome and Itoh [NAK91] , and Hidaka, Matsuda, and Asakura [HID90] discuss the potential for improvements in DRAM and memory interface technology.

Most of the related work in software approaches has focused on ways to reduce the memory bandwidth requirements of a program. For example, there is a plethora of research describing algorithms for register allocation, one of the fundamental transformations for reducing a program's memory bandwidth requirements. Register allocation identifies variables that can be held in registers. By allocating the variable to a register, memory loads and stores previously necessary to access the variable can be eliminated. An evaluation of the register coloring approach to register allocation showed that up to 75 percent of the scalar memory references can be removed [CHO90] using these techniques.

Another class of code transformation that can reduce a program's memory bandwidth requirements are blocking transformations such as cache blocking and register blocking. These transformations can profitably be applied to codes that process large sets of data held in arrays. For example, consider the multiplication of two large arrays. By large, we mean that the arrays are much larger than the size of the machine's data cache. Cache blocking attempts to restructure the code that processes the arrays so that rather than processing entire rows or columns, sub-arrays of the arrays are processed.

Because the arrays are larger than the cache, processing the entire array results in data being read from memory to the cache many times. Cache blocking, however, transforms the code so that a block of the array that will fit in the cache is read in once, used many times, and then replaced by the next block. The performance benefits from this transformation can be quite good. Lam, Rothberg, and Wolf [LAM91] show that for multiplication of large, floating-point arrays, cache blocking can easily triple the performance of a cache-based system.

Register blocking is similar in concept to cache blocking, but instead of transforming code to reduce the number of redundant loads of array elements into the cache, register blocking transforms code so that unnecessary loads of array elements are eliminated altogether. Register blocking can be considered a specific application of scalar replacement of subscripted variables [CAL90] and loop unrolling. Scalar replacement identifies subscripted variables which appear in loops but whose subscripts are loop invariant. These variables can be loaded into registers outside the loop allowing register references replace the memory references inside the loop. Unrolling the loop exposes a block of these subscripted variables to which scalar replacement can be applied. Register blocking and cache blocking can be used in combination to reduce the number of memory references and cache misses.

Another program transformation that reduces a program's memory bandwidth requirements is called recurrence detection and optimization [BEN91] . Knuth [KNU73] defines a recurrence relation as a rule which defines each element of a sequence in terms of the preceding elements. Recurrence relations appear in the solutions to a large number of compute and memory intensive problems. Interestingly, codes containing recurrences often cannot be vectorized. Consider the following C code:

```
for (i = 2; i < n; i++)
   x[i] = z[i]* (y[i] - x[i-1]);
```

This is the fifth Livermore loop, which is a tri-diagonal elimination below the diagonal. It contains a recurrence since `x[i]` is defined in terms of `x[i-1]`. By detecting the fact that a recurrence is being evaluated, code can be generated so that the `x[i]` computed on one iteration of loop is held in a register and is obtained from that register on the next iteration of the loop. For this loop, the transformation yields code that saves one memory reference per loop iteration.

2: Wide-Bus Machines (WBMs)

We define a wide-bus machine (WBM) to be an architecture where the implementation has a memory bus width that is at least twice the size of the integer data type handled by the processor and assumed by the programmer. For emerging technology, integers are 32 bits, and WBMs have a bus width of 64 bits. We will assume these parameters for the rest of the paper.

A WBM may or may not support 64-bit operations on the data, although we expect most WBMs will. Note that an instruction-set architecture may contain operations on 64-bit data, but a particular implementation may not be a WBM. For example, the VAX-11 instruction set contained instructions for storing quad-words (64 bits) and octa-words (128 bits), but the SBI (Synchronous Backplane Interconnect) on the VAX-11/780 was only 32 bits wide [VAX82] .

We are aware of at least three WBMs available from system vendors, the DEC Alpha [ALP92] , the Motorola 88110 [MC891] , and the MIPS R4000 [KAN92] . These machines share the characteristic that 64-bits can be loaded from or stored to memory. Of these machines, the DEC

Alpha is closest to our vision of what future 64-bit machines will look like. The MIPS R4000 and Motorola 88110 are limited in their ability to manipulate 64-bit fixed-point data. The DEC Alpha, on the other hand, can load 64 bits of data to either the fixed-point registers or floating-point registers. Indeed, there are no load or store instructions for byte and 16-bit data. Furthermore, the DEC Alpha provides arithmetic operations on 64-bit fixed-point data†.

We expect as the technology improves and manufacturing costs drop, implementations of WBMs will become widely available. In order to experiment with and evaluate code improvements that are designed to take advantage of the memory bandwidth provided by WBMs, we designed an instruction-set architecture that included 64-bit load and store operations. Our design reflects what we believe the first generation of many WBMs will look like. We expect that as technology improves WBMs will evolve much like 32-bit architectures have.

Our prototype WBM is based on the MIPS R3000 instruction-set architecture with four additional instructions for manipulating 64-bit data. The machine has 32 general-purpose, 32-bit fixed-point registers and 16 floating-point registers that may hold either 32 or 64 bits (typically either IEEE floating-point standard single- or double-precision numbers). The instruction set includes new operations for loading and storing 64-bits of data to/from the fixed-point registers. When 64-bit data is loaded into the fixed-point registers, it is placed in a even-odd register pair. Similarly, when 64-bit data is stored from the fixed-point registers, the data in an even-odd register pair is written to memory.

For the floating-point unit, the existing MIPS R3000 architecture contains a load double-precision instruction that fetches 64 bits from memory and puts it in a floating-point register. Similarly, a store double instruction is available. We added a load single-precision double instruction that loads two single-precision floating-point values from consecutive memory locations into a consecutive floating-point register pair. An analogous store single-precision double instruction was also added. Similar to many architectures, loads and stores of 64-bit data items must be *naturally aligned*. By naturally aligned, we mean that data that is 2^N bytes in size is stored at an address that is a multiple of 2^N. For example, a 64-bit item must be stored in memory at an address where the lower 3 bits are zero.

In summary, our machine is a super-set of the MIPS R3000 instruction-set architecture with four additional instructions for loading and storing 64-bits of data. The extensions are shown below using RTLs (Register Transfer Lists) [DAV84] and the corresponding assembly language instructions:††

```
RTL Representation                          Assembly Lang.
r[4]=R[r[29]+8];r[5]=R[r[29]+12];           ld64 %4,8(%29)
R[r[29]+8]=r[4];R[r[29]+12]=r[5];           st64 %4,8(%29)
f[0]=F[r[29]+16];f[2]=F[r[29]+20];          ldf64 %0,16(%29)
F[r[29]+16]=f[2];F[r[29]+20]=f[4];          stf64 %2,16(%29)
```

It is assumed that the implementation of the instruction-set architecture will support the above operations in a single memory operation.

The above extensions represent what a first-generation WBM might look like. Certainly as VLSI design and fabrication technology improves, one can expect that additional 64-bit operations and features could be included in the instruction-set architecture. The above extensions, in some sense, represent a minimal implementation and thus will serve to give a lower bound on the performance improvements provided by a wide bus.

3: Memory Bandwidth Code Improvements

In order to design, implement, and evaluate memory bandwidth code improvements for WBMs, we modified an existing retargetable optimizer to include the memory bandwidth code improvements. The optimizer, called *vpo*, is a sophisticated, global optimizer that can be used to build optimizing compilers [BEN88, DAV84, DAV84] . Figure 1 contains a schematic showing the overall organization of a C compiler constructed using *vpo*. Vertical columns represent logical phases which operate serially. Columns that are divided horizontally into rows indicate that the sub-phases of the column many be executed in an arbitrary order. For example, instruction selection may be performed at any time during the optimization process. Global data-flow analysis, on the other hand, is done after the basic block optimizations, instruction selection, and evaluation order determination, but before local register assignment, common subexpression elimination, etc.

This structure makes *vpo* ideal for evaluating the effectiveness of new code improvements. First, it provides a framework for easily adding new code improvement algorithms. Second, it includes most code improvements that are found in production optimizing compilers. Third, phase-ordering problems that can interfere with the effectiveness of optimizations are largely eliminated.

†Programmers will still treat integers (e.g., C's *int* type and Fortran's *integer*) as 32-bits initially.

††In this paper, all machine code will be shown using RTLs.

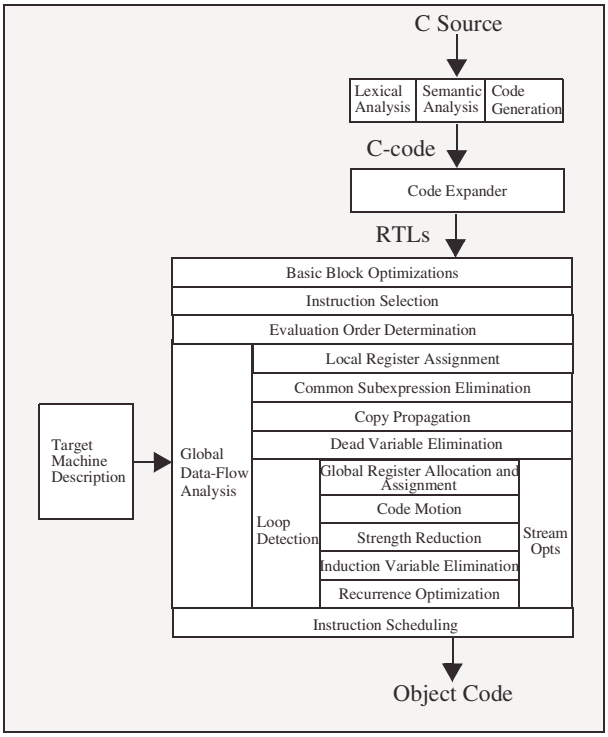

Figure 1. Schematic of vpo-based C compiler.

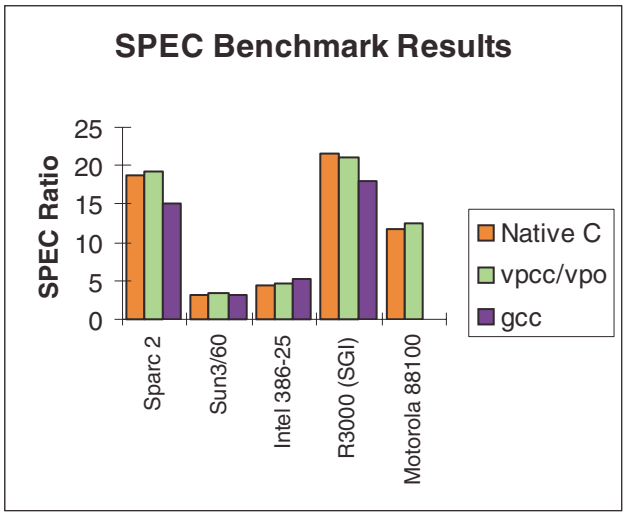

Figure 2. C Compiler Performance Comparison.

The graph in Figure 2 compares the performance of C compilers constructed using *vpo* and other production C compilers on six widely used machines. The results are from the execution of the C component of the SPEC benchmark suite. The table shows that *vpo* produces code that compares favorably to that produced by production compilers. Consequently, we expect that the performance benefits reported here will be close to what other optimizing compilers can expect to realize. Third, *vpo* and a companion tool, called *ease* (Environment for Architecture Study and Experimentation [DAV90], provide facilities for experimenting and evaluating the performance of new architectural features. These facilities and how they are used for this study are described in Section 4.

Within *vpo*'s framework, we implemented a C compiler for the architecture described previously. This C compiler utilized the new 64-bit instructions in three ways. Two of the uses were easy to implement, and we expect that most compilers could include these uses with minor modifications. One use of the additional instructions that was simple to add was to save and restore the volatile fixed-point registers at function entry and exit (i.e., as part of the procedure prologue and epilogue code). Like many architectures, the MIPS R3000 function call/return interface requires that the called routine save the contents of certain registers if it uses them. For the MIPS architecture, the contents of fixed-point registers 16 through 23 must be preserved across calls. A second straightforward use of the new instructions was the manipulation of C structures (i.e., structure assignment and passing arguments that are structures).

The optimizer, *vpo*, was modified to locate opportunities for a third, beneficial use of the 64-bit instructions and to apply code transformations so the available memory bus bandwidth could be better utilized. Consider the following C loop:

```
for (i = 0; i < n; i++)
    dy[i] = dy[i] + da*dx[i];
```

This is the *daxpy/saxpy* loop from the well known benchmark program *linpack*. Fifty-six percent of *linpack's* execution time is spent in this loop. Each iteration of this loop reads two array elements and writes one. On a WBM, when the arrays are single-precision (*saxpy*), 50 percent of the available memory bandwidth is not utilized. Each load and the store move just 32 bits of data. Using several restructuring techniques, *vpo* can produce machine code that fully utilizes the available memory bandwidth for each access to the arrays.

After a loop has been identified as a candidate for transformation, the first step is to unroll the loop so that there are *buswidth/elementwidth* consecutive accesses to each array in the loop. On a machine with a 64-bit bus, where

```
/* Incoming Register Assignments
 * r[4] cotains n
 * r[6] contains da
 */
  f[0]=r[6]          /* copy da to float reg
  r[8]=0;            /* initialize i
  r[2]=r[4]/2;       /* compute n/2
  r[6]=R[r[29]+dx];  /* load address of dx
  r[7]=R[r[29]+dy];  /* load address of dy
  PC=0‘r[2],L1;      /* test n/2
  r[11]=r[8]+r[7];   /* compute address of dy[1]
  r[11]=r[11]+4;
  r[12]=r[8]+r[6];   /* compute address of dx[1]
  r[12]=r[12]+4;
  r[9]=r[2]{2;       /* compute ending address for
  r[9]=r[9]+r[12];   /* testing loop termination
L2:
  f[2]=F[r[12]-4];   /* load dx[i]
  f[4]=f[0]*f[2];    /* compute da*dx[i]
  f[2]=F[r[11]-4];   /* load dy[i]
  f[2]=f[2]+f[4];    /* compute dy[i]+da*dx[i]
  F[r[11]-4]=f[2];   /* store dy[i];
  f[2]=F[r[12]];     /* load dx[i+1]
  f[4]=f[0]*f[2];    /* compute da*dx[i+1]
  f[2]=F[r[11]];     /* load dy[i+1]
  f[2]=f[2]+f[4];    /* comp. dy[i+1]+da*dx[i+1]
  F[r[11]]=f[2];     /* store dy[i+1]
  r[11]=r[11]+8;     /* advance to next pair of dy
  r[12]=r[12]+8;     /* advance to next pair of dx
  PC=r[12]<r[9],L2;  /* test if all elmts proc’ed
L1:
  PC=RT;             /* return
```

Figure 3. Saxpy after conventional optimizations.

```
/* Incoming Register Assignments
 * r[4] cotains n
 * r[6] contains da
 */
  f[0]=r[6]          /* copy da to float reg
  r[8]=0;            /* initialize i
  r[2]=r[4]/2;       /* compute n/2
  r[6]=R[r[29]+dx];  /* load address of dx
  r[7]=R[r[29]+dy];  /* load address of dy
  PC=0‘r[2],L1;      /* test n/2
  r[11]=r[8]+r[7];   /* compute address of dy[1]
  r[11]=r[11]+4;
  r[12]=r[8]+r[6];   /* compute address of dx[1]
  r[12]=r[12]+4;
  r[9]=r[2]{2;       /* compute ending address for
  r[9]=r[9]+r[12];   /* testing loop termination
L2:
          /* load dx[i] and dx[i+1] (double load)
  f[2]=F[r[12]-4];f[4]=F[r[12]];
  f[2]=f[0]*f[2];    /* compute da*dx[i]
  f[4]=f[0]*f[4];    /* compute da*dx[i+1]
         /* load dy[i] and dy[i+1] (double load)
  f[6]=F[r[11]-4];f[8]=F[r[11]];
  f[2]=f[6]+f[2];    /* compute dy[i]+da*dx[i]
  f[4]=f[8]+f[4];    /* comp. dy[i+1]+da*dx[i+1]
       /* store dy[i] and dy[i+1] (store double)
  F[r[11]-4]=f[2];F[r[11]]=f[4];
  r[11]=r[11]+8;     /* advance to next pair of dy
  r[12]=r[12]+8;     /* advance to next pair of dx
  PC=r[12]<r[9],L2;  /* test if all elmnts proc’ed
L1:
  PC=RT;             /* return
```

Figure 4. Saxpy using 64-bit load and stores.

single-precision data is 32 bits, the loop is unrolled once to yield the following code:

```
for (i = 0; i < n/2; i += 2) {
    dy[i] = dy[i] + da*dx[i];
    dy[i+1] = dy[i+1] + da*dx[i+1];
    }
```

After performing various other code improvements such as strength reduction, induction variable elimination, and code motion (see Figure 1), *vpo* produces the machine code shown in Figure 3.

The loop performs four loads and two stores each loop iteration. Furthermore, each load and store only uses half of the available memory bus bandwidth. However, it is now obvious that there are consecutive references to each of the array elements, and *vpo*, with modifications to locate the consecutive references, produces the code shown in Figure 4.†

†For simplicity of presentation, the code has not been scheduled to avoid pipeline conflicts.

The code in Figure 4 is a dramatic improvement over the code in Figure 3. First, the new code executes only 10 instructions per loop iteration, whereas the loop without the 64-bit loads and stores executes 13 instructions per loop iteration, a 23 percent reduction. Second, there are now only two loads and one store in the loop. Both loads and the store fully utilize the 64-bit bus.

Unfortunately, one characteristic of the machine complicates the implementation of the algorithms that perform the above transformations—all memory accesses must be naturally aligned. In order to utilize the 64-bit loads and stores to load or store two array elements on each iteration of the loop, the optimizer must be able to guarantee that the memory addresses are aligned on an eight-byte boundary. For many loops, *vpo* is able to perform the compile-time analysis to determine if the array references are aligned. For example, the starting address of singly-dimensioned arrays that are globals can be guaranteed to be aligned on a 64-bit boundary by the code generator. Similarly, for singly-dimensioned arrays that are locals, the

code generator and optimizer can cooperate to ensure that the starting address of the array on the stack is aligned.

Unfortunately, there are many instances where compile-time analysis cannot guarantee that the alignment requirements will be satisfied, and further, there are codes where alignment of a vector is not known until run-time. In fact, in the previous loop, the arrays `dx` and `dy` are parameters to the routine containing the loop. In the absence of interprocedural analysis, it is impossible, at compile-time, to determine if the arrays are aligned on an eight-byte boundary. Even when the starting address of an array can be determined at compile-time and it is determined to be aligned properly, processing the array two elements per loop iteration can result in unaligned accesses. Consider the following C code:

```
for (i = 0; i < DIM; i++) {
    for (k = 0; k < DIM; k++) {
        tmp = a[i][k];
        for (j = 0; j < DIM; j++)
            c[i][j] += tmp * b[k][j];
        }
    }
```

This is an implementation of the standard algorithm for multiplying two square arrays. Even when the starting addresses of the arrays are aligned properly and it is possible to determine this at compile-time, not all accesses to pairs of array elements may be aligned properly. Notice that the inner loop processes a row of `b`. For the C programming language, where arrays are stored in row-major order, whether the beginning address of each row of `b` is 64-bit aligned will depend on size of an element and the number of elements in the row. Assuming that the element size is 32-bits (single-precision floating-point numbers), if the number of elements in a row is even, the starting address of each row will be aligned. If, on the other hand, the number of elements in each row is odd, the starting address of every other row will be aligned. Similarly for FORTRAN, which stores arrays in column-major order, the accesses to the beginning of a column of an array may or may not be aligned.

In order to gain the performance benefits of full memory bus utilization even in the cases where alignment requirements cannot be insured at compile-time, *vpo* employs a technique called loop replication. In this technique, a loop where 64-bit loads and stores can profitably be used is replicated so that two instances of the loop exist. Code for one instance of the loop is generated assuming that the address of the first reference is aligned, and code for the other instance is generated assuming that the address of the first reference is not aligned. A test is inserted that checks the address of the first reference to determine if it is aligned or not. If it is aligned, the aligned loop will be executed, otherwise the non-aligned loop is executed. Figure 5 illustrates the transformation.

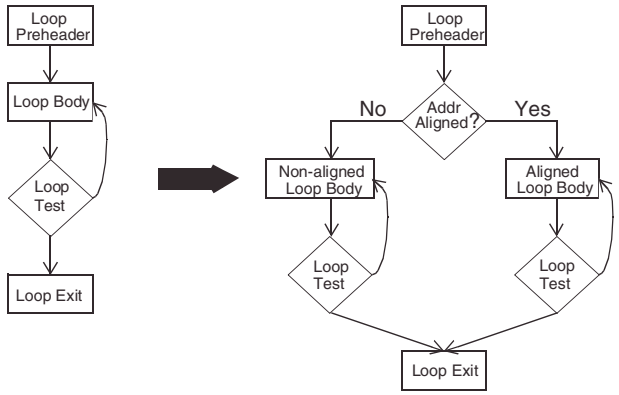

Figure 5. Loop replication transformation

Typically, the aligned loop will use 64-bit instructions to load or store the aligned elements, while the non-aligned loop will use 32-bit loads and stores to avoid alignment faults. However, this is not always the case. For some replicated loops, all versions of the loop can make use of the 64-bit data path. Consider the following C code:

```
for (k = 0; k < n; k += 3)
    D[k/3] = A[k] * A[k+1] / A[k+2];
```

Each loop iteration loads three consecutive elements of array `A`. If the compiler cannot determine whether the address of `A[0]` (the first reference) is aligned, the loop replication algorithm produces two instances of the loop. One assumes that the address of `A[0]` is aligned, and the other assumes it is not. However, both instances of the loop make use of the new 64-bit load instructions. One instance of the loop will load `A[k]` and `A[k+1]` using a 64-bit load, and `A[k+2]` using a 32-bit load. The other instance of the loop will load `A[k]` using a 32-bit load, and `A[k+1]` and `A[k+2]` using a 64-bit load.

A loop that can benefit from 64-bit memory operations may need to be replicated more than once. Consider the previously shown matrix multiplication loop. After being unrolled once, the inner loop contains references to two arrays that are candidates for being accessed via the 64-bit loads and stores. Thus, there are four possible combinations: 1) neither array reference is aligned properly, 2) both array references are aligned properly, 3) `c` is aligned properly while `b` is not, and 4) `c` is not aligned properly while `b` is

aligned. In general, a loop may need to be replicated 2^N-1 times where N is the number of distinct memory references (i.e., different arrays or being referenced with a different induction variable) in the loop that are candidates for being accessed via the 64-bit loads and stores. Such loop replication can make the code larger. In practice, the loops that can benefit from loop replication are typically small. Furthermore, they typically process a small number of (usually no more than three) distinct arrays. Section 4 provides some data on how much larger loop replication makes code grow.

4: Evaluation

We implemented the above optimizations in an optimizing C compiler targeted to our prototype WBM. To evaluate the effectiveness of the code improvements, we used an environment called *ease* (Environment for Architecture Study and Experimentation [DAV90]. *ease* provides two capabilities that are necessary to perform the evaluation. First, *ease* provides the capability for emulating one architecture on another. Second, it provides the capability for gathering detailed dynamic measurements of a program's performance on an architecture.

ease uses the following technique to emulate a hypothetical architecture on a host machine. Normally, the last step of the optimizer, *vpo*, is to translate the RTLs produced for a target machine to assembly language for the target machine. The final stages of compilation runs the target machine's assembler and linker to produce an executable. For emulation, the RTLs are translated to sequences of host-machine instructions that emulate the target-machine instructions. Because our hypothetical target machine is a superset of the host machine (MIPS R3000-based Silicon Graphics Iris), only the four additional 64-bit instructions need to be emulated.

To gather detailed measurements of an architecture's dynamic behavior running a particular program, after code is generated and optimization is complete, *ease* inserts code to count the number of times instructions are executed. Information about the instructions used by the program is written to a file. When the program is run to completion, the count information is written out and correlated with the information about the instructions. For this study, *ease* measured, broken down by type, the number of memory references performed and the number of instructions executed. The interested reader is referred to WHA90 and DAV90 for more details on how *ease* efficiently and simply gathers these detailed measurements.

Set 1 (Typical User Code)

Program	*Description*
eqntott	SPEC benchmark program
espresso	SPEC benchmark program
li	SPEC benchmark program
gcc	SPEC benchmark program
grep	Unix utility (regular exp. search)
nroff	Unix utility (text formatting)
yacc	Unix utility (parser generator)
compact	Unix utility (file compaction)
diff	Unix utility (file difference)

Set 2 (Memory Intensive Kernels)

Program	*Description*
saxpy	Linpack program
First diff	Livermore loop 12
Gauss	Gaussian elimination
matmut	Livermore loop 21
state	Livermore loop 7
hydro	Livermore loop 1
sobel	Graphics transform
laplace	Graphics transform

Table I. Programs used for evaluation.

To evaluate the effects of the code improvements on memory traffic, we used two sets of programs (see Table I). Set 1 consists of a group of well-known user programs and the four C programs from the original SPEC benchmark suite [SYS89] . These programs are intended to represent typical user code. Set 2 consists of a set of loops that represent memory intensive kernels. For example, several of the loops are from the well known Livermore loops benchmark suite. Using the typical user code and *ease*, the effectiveness of using the 64-bit instructions to save and restore registers at function entry and exit, to copy and move structures, and to fetch and store pairs of elements of arrays was measured. Figure 6 shows, broken down into two categories, the percentage reduction in memory operations performed and the number of instructions executed for Set 1. The categories are the savings resulting from the straightforward modification of the compiler to use the 64-bit instructions (e.g., saving and restoring registers at function entry and exit, and manipulating structures when possible), and the savings resulting from the loop code improvements.

gcc and *nroff* showed a 12 and 13 percent reduction in memory references using the new 64-bit loads and stores. Figure 6 reveals that most all of the savings comes from using the new 64-bit instructions to save and restore registers at function entry and exit and to manipulate structures. Only

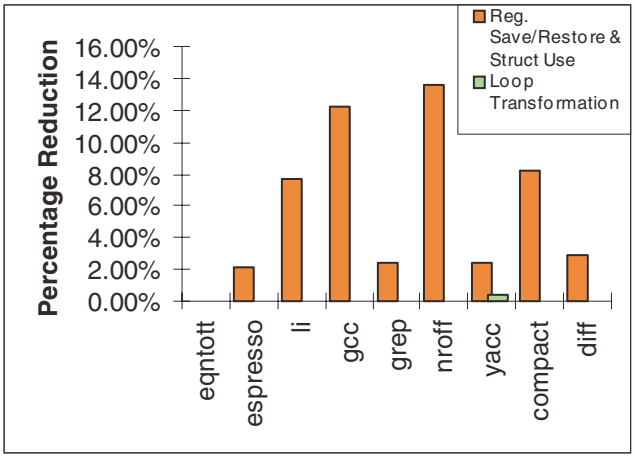

Figure 6. Reduction in memory operations using

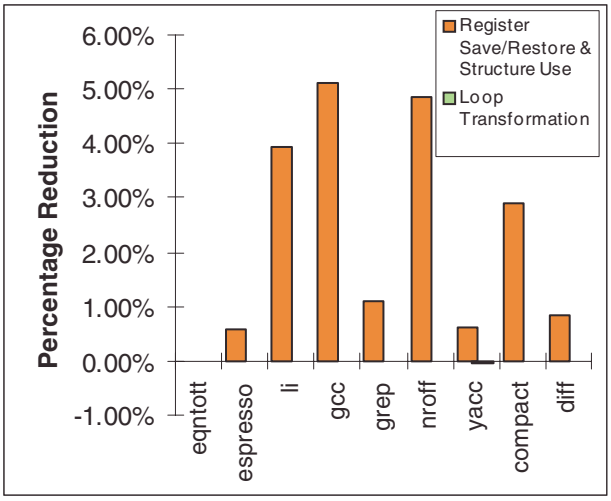

Figure 7. Reduction in instructions executed using the 64-bit load and store instructions

yacc showed a perceptible use of the 64-bit instructions to process arrays of data. For the nine programs, average savings overall was about six percent.

Figure 7 shows the reduction in instructions executed for Set 1. Here the average savings was only 2.2 percent. Using the instructions to save and restore registers at function entry and exit and to manipulate structures was the source of most all the savings. Indeed, for two programs, *espresso* and *yacc*, the loop code transformation resulted in more instructions being executed. The reason for this is that the loop code transformation inserts code to test alignment of a data structure. In the case where the reference is not aligned on a 64-bit boundary, the original code is executed. Consequently, the modified program executes more instructions than the original. This highlights the importance of keeping data structures properly aligned if possible.

For the kernel loops, as one would expect, the savings are much greater. Figure 8 shows the percentage reduction in memory references and instructions executed. For these loops, all the improvement is due to the loop code transformation. For a couple of the loops, the number of memory references was reduced by nearly 50 percent. The average for all the loops was 28 percent. For instructions executed, the average reduction was 10 percent. For these memory- and compute-intensive loops, the loop transformation clearly can provide significant performance benefits, both in reducing the number of memory references (and hence increasing bus utilization) and the number of instructions executed.

One disadvantage of loop replication to handle misaligned accesses is that it increases the size of the code. Figure 9 shows the increase in the size of the code for the kernel loops. The largest increase is 800 percent. This increase may seem large, but these kernel loops are typically no more than ten to fifteen statements in length, and they are usually embedded in much larger applications. Consequently, the code growth seems reasonable given the potential performance gains for these kernels. Of more interest, however, is the effect the code growth could have on instruction cache performance. Many processors include instruction caches and better performance can be obtained if a loop fits in the instruction cache. A transformation that increases the size of a loop such that it no longer fits in the instruction cache can result in poor performance. While code replication increases code size, the size of the inner (and important) loop does not grow. The extra code for testing alignment is placed outside the inner loop. In fact, the size of the inner loop may shrink significantly when the 64-bit loads and stores are employed. Thus, with the code improvement, a inner loop that previously might not have fit in the instruction cache may now fit.

5: Other Improvements

While the code improvements just described provide significant performance benefits, there are additional enhancements that should provide even better performance. The code improvement just described attempts to replace 32-bit memory references with more efficient 64-bit memory references. The same techniques can be applied to 8-bit and 16-bit memory references. Consider a loop that is processing an array of pixels (8-bit data). By unrolling this loop eight times, eight pixels could be loaded and processed each loop

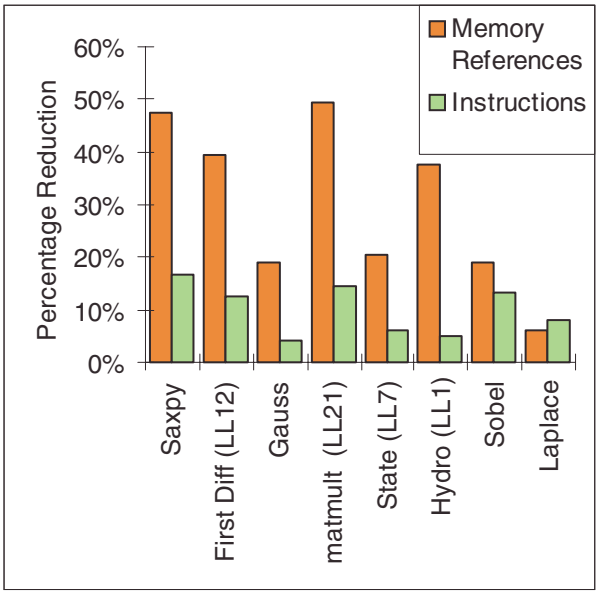

Figure 8. Reduction in memory references and instructions executed for kernel loops.

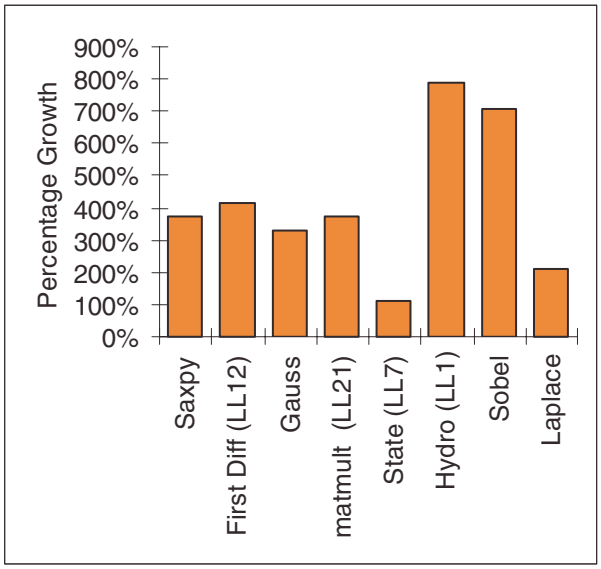

Figure 9. Code size growth due to loop replication.

iteration. Of course, unpacking the pixels, operating on them, and then packing them back into the register will take some additional cycles, the net effect should be a reduction in the number of cycles.

The above code improvement, if widely applicable (and we believe it is), points to the need for additional support in the instruction set. For example, it may prove advantageous to include instructions for unpacking data from a register into a series of registers, and for packing data back into a register. We plan to explore the effects of these architectural changes.

The code improvements described thus far apply to codes that process arrays of data. Similar code improvements could also be applied to scalar data. The following code contains access to two different scalar variables in close proximity to one another (in this case, in the same expression).

```
x = v1 * v2;
```

If the two scalars, `v1` and `v2`, were not allocated to registers, they would need to be loaded into registers before the multiplication operation could be performed. By analyzing the loads of scalar data, and finding pairs that are loaded near each other, the optimizer could ensure that the pair is allocated contiguously in memory and that they are aligned properly. They could then be accessed using a load instruction that fetches both.

6: Summary

Sixty-four bit microprocessor architectures and their implementations are the next logical step in the evolution of microprocessor design and implementation. Indeed, several vendors are beginning to ship the first 64-bit chips. This paper has described and evaluated, for a prototypical 64-bit architecture, some code transformations that take advantage of the ability to load and store 64-bits in a single memory operation. For typical user code, the number of memory references performed by a program can be reduced by an average of six percent, with some programs yielding savings of 10 to 15 percent. For compute- and memory-intensive programs, the savings can run as high as 45 percent. We believe that as 64-bit architectures mature and become more readily available, the code improvements described here and others will become standard, necessary components of high-performance optimizing compilers for 64-bit architectures.

7: Acknowledgements

David Whalley developed the original *ease* tool. Ricky Benitez provided invaluable assistance with the modification of *vpo*.

8: References

[ALP92] *Alpha Architecture Handbook*. Digital Equipment Corporation, 1992.

[BEN88] Benitez, M. E. and Davidson, J. W. A Portable Global Optimizer and Linker. *Proceedings of the SIGPLAN '88 Symposium on Programming Language Desig*n, pages 329-338, Atlanta, GA, June, 1988.

[BEN91] Benitez, M. E. and Davidson, J. W. Code Generation for Streaming: an Access/Execute Mechanism. *Proceedings of the Fourth International Conference on Architectural Support*, pages 132-141, Santa, CA, April, 1991.

[CAL90] Callahan, D. and Carr, S. and Kennedy, K. Improving Register Allocation for Subscripted Variables. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 53-65, White Plains, NY, June, 1990.

[BUR92] Bursky, D. Memory-CPU Interface Speeds up Data Transfers. *Electronic Design* 40(6):137-142 March 1992.

[CHO90] Chow, F. C. and Hennessy, J. L. The Priority-Based Coloring Approach to Register Allocation. *ACM Transactions on Programming Languages and Systems* 12(4):501-536 October 1990.

[DAV84] Davidson, J. W. and Fraser, C. W. Code Selection through Object Code Optimization. *Transactions on Programming Languages and Systems* 6(4):7-32 October 1984.

[DAV90] Davidson, J. W. and Whalley, D. B. Ease: An Environment for Architecture Study and Experimentation. *Proceedings of the 1990 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 259-260, Boulder, CO, May, 1990.

[DAV84] Davidson, J. W. and Fraser, C. W. Register Allocation and Exhaustive Peephole Optimization. *Software--Practice and Experience* 14(9):857-866 September 1984.

[GOO83] Goodman, J. R. Using Cache Memory to Reduce Processor-Memory Traffic. *Proceedings of the 10th Annual Symposium on Computer Architecture*, pages 124-131, Stockholm, Sweden, June, 1983.

[HEN90] Hennessy, J. L. and Patterson, D. A. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, 1990.

[HID90] Hidaka, H. and Matsuda, Y. and Asakura, M. The Cache DRAM Architecture: a DRAM with an on-chip cache memory. *IEEE Micro* 10(2):14-25 April 1990.

[KAN92] Kane, G. and Heinrich, J. *MIPS RISC Architecture*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[KNU73] Knuth, D. E. Volume 1: *Fundamental Algorithms*. Addison-Wesley, Reading, MA, 1973.

[LAM91] Lam, M. and Rothberg, E. E. and Wolf, M. E. The Cache Performance and Optimizations of Blocked Algorithms. *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63-74, Santa Clara, CA, April, 1991.

[MC891] *MC88110: Second Generation RISC Microprocessor User's Manual*. Motorola, Inc., Phoenix, AZ, 1991.

[NAK91] Nakagome, Y. and Itoh, K. Reviews and Prospects of DRAM Technology. *IEICE Transactions on Communications Electronics Information and Systems* 74(4):799-811 April 1991.

[SYS89] Systems Performance Evaluation Cooperative. *SPEC Newsletter: Benchmark Results* Fall 1989.

[VAX82] *VAX Hardware Handbook*. Digital Equipment Corporation, Maynard, MA, 1982.

[WHA90] Whalley, D. B. *Ease: An Environment for Architecture Study and Experimentation*. Ph.D. Dissertation, University of Virginia, Charlottesville, VA, 1990.