Program Balance and its Impact on High Performance RISC Architectures*

Lizy Kurian John and Vinod Reddy
Computer Science and Engineering
University of South Florida
Tampa, FL 33620

Paul T. Hulina and Lee D. Coraor
Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802

Abstract

Information on the behavior of programs is essential for deciding the number and nature of functional units in high performance architectures. In this paper, we present studies on the balance of access and computation tasks on a typical RISC architecture, the MIPS. The MIPS programs are analyzed to find the demands they place on the memory system and the floating point or integer computation units. A balance metric that indicates the match of accessing power to computation power is calculated. It is observed that many of the SPEC floating point programs and kernels from supercomputing applications typically considered as computation intensive programs, place extensive demands on the memory system in terms of memory bandwidth. Access related instructions are seen to dominate most instruction streams. We discuss how these instruction stream characteristics can limit the instruction issue in superscalar processors. The properties of the dynamic instruction mix are used to alert computer architects to the importance of memory bandwidth. Single instruction stream parallelism will not be much greater than two if memory bandwidth is only one. A decoupled access/execute architecture with multiple load/store units and queues which alleviate the balance problem is presented.

Keywords: Access/Execute Balance, Memory Bandwidth, Pipeline Balance, Program Behavior.

1 Introduction

Increasingly sophisticated circuit design and well-thought-out architectures have lifted microprocessors to new pinnacles of performance. In all high-performance architectures, there should be a close match between the number and nature of functional units in the system and the dynamic instruction mix of the applications running on the system. The functional units on a machine must reflect the instruction frequency if we are to achieve efficient utilization of functional units. In this paper we study MIPS programs from DECStation 5000 to obtain a picture of the balance of accessing and computation, while executing typical floating point programs.

A computer is said to be balanced when it can operate in a steady state manner with both memory accesses and floating point operations being performed at peak speed [3]. Callahan, Cocke and Kennedy [3] defined two metrics called 'machine balance' and 'loop balance' which together indicate how efficiently a loop can be executed on a particular pipelined processor. They defined machine balance β_m as the rate at which floating point operands can be fetched from memory compared to the rate at which floating point operations can be performed. They also defined a metric called loop balance β_L, which is naturally associated with each program loop. Loop balance is defined as the ratio of the number of words accessed to the number of flops performed. As evident from the definitions, when machine balance is less than loop balance, the loop needs more data than what the fetch unit can provide and such programs are traditionally called memory-bound programs. If loop balance is less than machine balance, the loop is compute-bound and if the two metrics are equal, the loop is said to be balanced. They further noticed that in pipelined architectures, the equation for loop balance has to be adjusted for idle cycles due to pipeline interlock. To perform an experimental evaluation of the balance of typical floating point programs on a typical RISC architecture is one of the objectives of this study.

Many architectures perform accessing and address arithmetic concurrently with floating point operations to construct a fast processor [3]. A typical example is the decoupled access/execute (DAE) architecture [22] [26] [11] [12]. To investigate the balance of conventional DAE architectures and design a balanced DAE configuration is another objective of this research effort.

Hammerstrom [8] studied memory reference entropy or information content of the memory references performed by a computer program. He studied CPU memory referencing behavior by separating the memory accesses that are used for fetching data elements required for computation purposes and those used for fetching data required for address generation purposes. He called the accesses solely used for address generation as *overhead memory references.* Examples of overhead memory references are accessing of loop indices and loop bounds. Hammerstrom observed that more than half of all memory references made by IBM 360 programs are overhead memory accesses. In some of the programs he traced, almost 90% of all memory references constituted an overhead. With the advent of RISC architectures with large register sets, most of the information required to compute addresses are allocated to registers. To the best of our knowledge, no program behavior study after the advent of RISC architectures and the RISC optimizing compilers, has done a study on the overhead memory accesses made by RISC programs. To perform such a study is another objective of this paper.

1.1 Overview

In this paper, we present an experimental evaluation of the balance and memory reference entropy of several floating point benchmark programs in a typical pipelined RISC processor pair, the MIPS R3000/3010. Several benchmark programs are executed on the DEC 5000 workstation and program instruction mix analyzed to investigate the loop or program balance. In section 2, we present analysis based on the dynamic instruction mix of the programs. Program balance is analyzed in section 2.1. Address arithmetic is quantified in section 2.2 and the access related part of the instruction mix is quantified in section 2.3. Overhead memory accesses are investigated in section 2.4 and some effects of loop transformations and compiler optimizations are discussed in section 2.5. In

*This work was supported in part by the National Science Foundation under grant number MIP–8912455.

section 3, we discuss the impact of the imbalance in the instruction stream on superscalar instruction issue, on the performance of DAE and other fine-grain parallel architectures, and on prefetching and latency reducing techniques. The design of a DAE architecture with multiple load/store units and multiple queues is also presented. We conclude the paper in section 4.

2 Instruction Stream Characterization

Our objective is to characterize floating point programs that typically constitute supercomputing applications. The experiments are performed on a DEC5000 workstation, which uses the MIPS R3000/3010 processors. The benchmark programs include five of the six floating point programs from the SPEC suite and several kernels from typical supercomputing applications including concatenated Lawrence Livermore Loops, convolution and correlation algorithms and the saxpy routine from Linpacks. The Lawrence Loops are concatenated together to yield a longer program. They are coded in Fortran and C; the Fortran version is labeled concat.f and the C version is called concat.c. The C programs are compiled with MIPS compilers CC 2.1. The FORTRAN benchmarks are compiled with the DEC FORTRAN compiler version 3.0. While compiling the SPEC programs, the SPEC Makefiles are strictly adhered to. The optimizations and various flags are in accordance with the SPEC Makefiles. The small benchmarks in C were compiled with the highest level of optimization, -O4 and also with the default optimization level of -O1. (Occasionally we refer to the -O1 and -O4 versions as unoptimized and optimized versions respectively.)

Two sets of tools are used to collect dynamic instruction statistics - (i) the MIPS profiling tools *pixie* and *pixstats* and (ii) in-house profiling tools that we developed. The profiling tools *pixie* and *pixstats* provide a variety of details connected with program execution - such as each opcode and its frequency of usage, a histogram of branch distances, floating point interlock statistics, etc. In addition to details furnished by *pixie* and *pixstats*, we were interested in obtaining information on address arithmetic and overhead memory references performed by the programs. This motivated us to develop our own profiling tools, which analyzed the instruction stream and tagged address arithmetic and overhead memory references. The arithmetic instructions used for computing addresses for memory accesses are called address arithmetic instructions. Address arithmetic instructions can be identified by back-tracking the base registers in load and store instructions. Manipulation of loop indices to point to the next required array element is also considered as address arithmetic. The memory accesses that fetch the operands used in the computation instructions are tagged as data access and memory accesses to fetch loop indices or such information used for address generation are tagged as overhead memory accesses. The profiler program accepts the *a.out* files of the benchmarks. Appendix I provides an example of how our profiling program would tag a sample program with data access, execute, address arithmetic, overhead memory access or branch tags.

2.1 Program Balance

We define program balance β on an architecture as the ratio of number of accesses performed to the number of floating point operations performed. i. e.

$$\beta = \frac{L+S}{F} \tag{1}$$

where L denotes the number of loads and S denotes the number of stores and F represents the number of floating point operations performed. Callahan et al. discusses how idle cycles in the floating point unit due to pipeline interlocks make the aforementioned metric definition inaccurate. Considering the pipeline interlocks, the balance metric gets modified to

$$\beta_{mod} = \frac{L+S}{F+I} \tag{2}$$

where I represents the number of idle cycles due to interlock. The *pixstats* output gives information about the number of floating point interlocks per flop. This lets us to compute a 'modified program balance', taking into account the pipeline balance. Since I is dependent on the implementation, the modified balance metric β_{mod} is valid only for a specific implementation of an architecture. Table 1 illustrates the percentage of loads and stores and floating point operations in the dynamic instruction mix of the programs we analyzed.

Benchmark	%Load + %Store	%fl pt	β	inter-locks per flop	β_{mod}
tomcatv	55.9	32.2	1.74	0.264	1.37
nasa7	52.0	24.0	2.17	0.614	1.35
fpppp	60.5	24.2	2.50	0.542	1.62
doduc	35.9	22.9	1.57	1.490	0.62
matrix300	55.5	18.4	3.02	0.125	2.68
G.M.	51.2	24.0	2.14	0.439	1.38

Table 1.a Balance in the SPEC Floating Point Programs

Benchmark	%Load + %Store	%fl pt	β	inter-locks per flop	β_{mod}
concat.c	55.2	30.1	1.83	0.71	1.07
concat.f	67.5	25.7	2.62	0.27	2.07
conv	66.0	16.2	4.07	0.27	3.20
corr	41.1	21.0	1.96	0.01	1.94
saxpy.eq	69.2	22.7	3.04	0.25	2.43
saxpy.uneq	35.2	11.5	3.06	0.001	3.06
G.M.	53.9	20.2	2.66	0.07	2.16

Table 1.b Balance in the small benchmark programs

According to our definition, if program balance is greater than 1, the program is access bound and if it is less than 1, it is compute bound. The metric β_{mod} is equivalent to the ratio of β_L and β_m in [3]. On the MIPS system, the SPEC floating point benchmarks indicate a balance of 2.14 (without considering idle cycles due to pipeline interlocks) and 1.38 considering the interlocks. The small benchmarks yield a balance of 2.66 without considering pipeline interlocks and 2.16 taking into effect the interlocks. Thus we observe that the SPEC floating point programs as well as the small benchmarks place more demands on the memory subsystem than on the floating point units in the MIPS environment. This is evident from Fig. 1 which illustrates the memory access part of the workload in comparison to the floating point computation workload.

The most surprising observation from Table 1 is that almost half of all the instructions in the instruction stream are memory references. We looked at reported instruction profiling studies on the MIPS architecture to verify that we were not making an error. In Cmelik et al.'s studies [4], for the SPEC floating point benchmarks, 46% of all instructions on the MIPS are memory reference instructions. In Sohi and Franklin's studies presented in [20], for the three SPEC floating point benchmarks they used, loads and stores add up to 37%. (It may be noted that

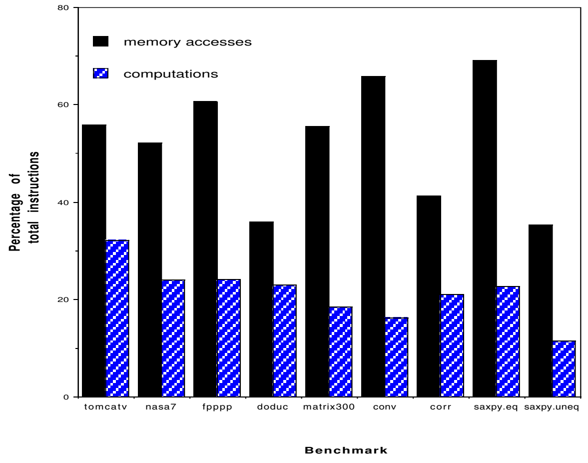

Figure 1: Number of memory access vs floating point computation instructions in typical benchmark programs

there is no difference in the order of the numbers, and that there is nothing that makes our findings suspect.)

Caches can capture locality and hence the number of words accessed from main memory may not be as high as the number projected by the instruction mix. But this number does denote the number of accesses required from the cache even assuming all hits. If the cache can provide only one word per cycle, we can safely conclude that all the SPEC floating point benchmarks are cache bound.

2.2 Address Arithmetic

Traditional CISC architectures often have fewer on-chip registers and for structured array references, an explicit address calculation with the basic array access formula is performed each time an array element is referenced. With the advent of RISC architectures, several registers are available to store base addresses for each of the vectors being accessed, and that could possibly reduce the address arithmetic involved. We analyzed MIPS code and observed that even for RISC architectures, in the unoptimized version, the array address is computed within each iteration. In the optimized case, the starting address is calculated once before entering the inner loop. The inner loop consists of only increments to the base address. Hence we may say that for iterative computations with nested loops, the address arithmetic involved in the unoptimized case is of the order $O(n^2)$ whereas in the optimized case, it is only of order $O(n)$. Table 2.a and Table 2.b illustrate the address arithmetic instructions for levels of optimization -O4 and -O1, expressed as a percentage of all instructions in the respective dynamic instruction stream. The results are obtained from our profiler program. The absolute number of instructions for each case is illustrated in Appendix II. It may be observed that for the best optimization level available, 13% of all instructions for the SPEC floating point benchmarks and approximately 11% of all instructions for the small benchmarks are address arithmetic. Table 2.b also illustrates the effect of optimizations. In the unoptimized version, address arithmetic constitutes 33% of all instructions, whereas in the optimized code, it is only 11%. (Each is expressed as a percentage of the total instructions in the respective instruction stream.) The Fortran version of the Lawrence Livermore Loops have the lowest percentage of address calculation instructions. This is because, most of the Lawrence Loops involve only sequential referencing of arrays and such address patterns are efficiently optimized by the Fortran compiler. For the SPEC benchmarks, only the results from the optimized version are presented, since finding out the effect of optimization on the SPEC benchmarks involves altering the Makefiles provided by SPEC.

Benchmark	%Address Arithmetic
doduc	17.1
nasa7	18.2
matrix300	21.3
fppp	9.2
tomcatv	6.2
G.M.	13.0

Table 2.a Address Arithmetic Instructions in the SPEC benchmarks expressed as a percentage of all instructions

Benchmark	Default (-O1)	Optimized (-O4)
concat.c	46.94	10.67
concat.f	30.65	3.90
conv	34.23	10.06
corr	21.88	18.66
saxpy.eq	26.43	3.97
saxpy.uneq	49.92	51.18
G.M.	33.53	10.80

Table 2.b Address Arithmetic Instructions in the small benchmarks expressed as a percentage of all instructions

2.3 Access-related Instructions

Many architectures now support concurrent execution of floating point computations and access-related operations. It is possible for load and store operations to be completely overlapped with floating point arithmetic operations. Load, store and address arithmetic constitute access related operations that may be performed in overlap with floating point computations. Table 3 presents the access related instructions in the instruction stream. We may observe that approximately two-thirds of all instructions in the instruction stream are access related. Fig. 2 illustrates the access-related operations as a part of the whole workload, and it may be seen that they dominate the instruction streams of all the benchmarks. The impact of such an unbalanced instruction mix can be tremendous but we postpone that discussion until section 3.

Benchmark	%Load	%Store	% Addr Arith	Total Access Related Instr(%)
doduc	26.6	9.3	17.1	53.0
nasa7	37.6	14.4	18.2	70.2
matrix300	37.0	18.5	21.3	76.8
fppp	43.8	16.7	9.2	69.7
tomcatv	43.6	12.3	6.2	62.1
concat.c	37.2	18.0	10.7	65.9
concat.f	47.9	19.6	3.9	71.4
conv	48.8	17.2	10.1	76.1
corr	30.6	10.5	18.7	59.8
saxpy.eq	45.9	23.3	4.0	73.2
saxpy.uneq	23.4	11.8	51.2	86.4
G.M.	37.4	15.0	11.8	68.9

Table 3 Access related instructions in the instruction stream

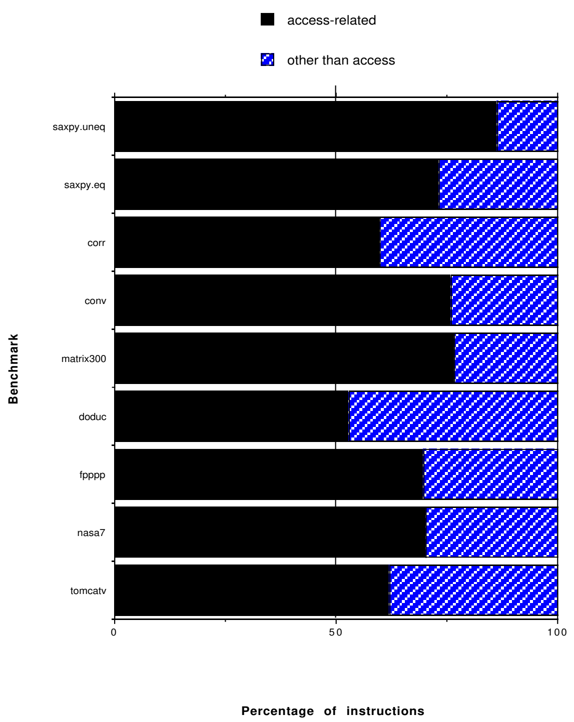

Figure 2: Access related instructions constitute the bulk of all instructions in typical benchmark programs

2.4 Overhead Memory References

As defined before, memory accesses that fetch data used solely for the generation of addresses are considered overhead memory accesses. Accessing of loop indices and bound variables belong to this category. In CISC code for iterative programs, often such variables are shuttled back and forth from memory in every iteration and that is the reason why Hammerstrom [8] found 90% of all memory accesses in some programs to be overhead. The corresponding behavior of RISC programs is investigated in this section. We observe that even in RISC code, in the -O1 version, index and address calculation information are shuttled back and forth from memory in every iteration. But in optimized code, a large part of such information for structured array accesses is stored in on-chip registers. Table 4.a illustrates the overhead memory references in the small benchmarks with compiler options -O1 and -O4, expressed as percentage of all instructions and of memory reference instructions alone. It may be observed that 36% of all memory references in the less optimized version (-O1) are overhead memory references. The overhead memory references constitute less than 1% in the optimized case (-O4), leading to the inference that RISC compilers are very successful in allocating address calculation information to on-chip registers and eliminating memory references required solely for generation of addresses. The overhead memory references in the SPEC benchmarks is also less than 1% and is illustrated in Table 4.b (absolute values and as percentage of memory references).

Benchmark	As percent of all instructions		As percent of memory ref instr	
	-O1	-O4	-O1	-O4
concat.c	12.01	0.44	31.47	0.79
concat.f	12.04	0.34	38.50	0.51
conv	19.81	0.66	39.72	0.97
corr	14.13	0.20	29.95	0.43
saxpy.eq	16.07	1.11	27.90	1.59
saxpy.uneq	20.10	0.58	55.80	1.59
G.M.	15.34	0.48	36.18	0.87

Table 4.a Overhead memory references in the small benchmarks

Benchmark	Total mem ref (in millions)	Overhead mem ref (in millions)	Overhead (%)
doduc	566.1	67.3	11.8
nasa7	4627.7	18.5	0.4
matrix300	1303.3	5.2	0.4
fpppp	1462.2	25.5	1.7
tomcatv	767.8	0.2	0.1
G.M.			0.8

Table 4.b. Memory Access Overhead in the SPEC benchmarks

2.5 Effects of Compiler Optimization

There are several loop transformations and optimizations that affect the balance of programs [3]. Common subexpression elimination and register saving of temporaries reduce the number of off-chip loads. In computations such as $x(i) = x(i-1) + y(i)$ and $x(i) = q + y(i) * (r * z(10) + t * z(11))$, register saving of temporaries can reduce loads. The MIPS compilers perform common subexpression elimination, elimination of RAW hazards by saving temporaries in registers etc, and hence the optimized results presented in this paper reflects the balance with these optimizations. The data in Appendix II illustrates a few trivial and widely understood effects of optimization. Some of these effects are reduction in program size, reduction in total number of memory references, reduction in address arithmetic, reduction in overhead memory references etc. Table 5 illustrates the effects quantitatively. The code size and the total number of memory references reduce to less than half with compiler optimization. The number of overhead memory references made in the optimized case is only 7% of that in the unoptimized (less optimized) case and the number of address arithmetic instructions in the optimized case is approximately 16% of that in the unoptimized case.

Characteristic	Optimized (-O4)/ default (-O1) (average)
Code Size	0.45
No of memory ref	0.48
Overhead memory ref	0.07
Address Arithmetic	0.16

Table 5. Some trivial effects of optimization

Another effect of compiler optimization is that it reduces the temporal locality in code. CISC architectures often had fewer on-chip registers and in those architectures and even in unoptimized RISC code, even scalars in iterative computations used to be stored in main memory. Repeated referencing of the scalars resulted in temporal locality in the code. The availability of more registers in RISC architectures resulted in scalars being allocated to registers. The temporal locality through scalar referencing thus got reduced. For loops with RAW dependencies, in CISC code, the results used to be written to memory (or cache) and then read again by the next iteration (with due care of RAW hazards). For RISC architectures with large register sets, the compilers allocate the intermediate results to on–chip registers and avoid referencing them from memory in further iterations. Conventional compiler optimization thus results in a lot of the temporal locality in the code to disappear. We do not imply that efficient register allocation is not good. Efficient register allocation significantly reduces the number of accesses performed by a program and is extremely beneficial. But *expectations from a data cache in optimized RISC programs should be formulated considering the fact that there is very little temporal locality in the code once compiler performs efficient register allocation.*

3 Impact of the Instruction Stream Behavior

The results reported in section 2 of this paper indicate surprising facts such as *half of all the instructions in the instruction stream are memory references, almost two thirds of all instructions are access related, etc. Such instruction mixes indicate fundamental bottlenecks that may have a serious impact on the design of future high performance architectures.* In section 3.1, we discuss the impact it can have on limiting the instruction issue in superscalar processors. In section 3.2, the impact on decoupled access/execute architectures is discussed in detail. A DAE architecture with multiple load/store units and multiple queues is presented. Section 3.3 shows the effect of a wider bus and higher bandwidth. In section 3.4, the impact of the instruction stream on latency reducing techniques is briefly discussed.

3.1 Superscalar Instruction Issue

Benchmark	Limit
tomcatv	1.79
dnasa7	1.92
fppp	1.65
doduc	2.78
matrix300	1.80
concat.c	1.81
concat.f	1.48
conv	1.52
corr	2.43
saxpy.eq	1.45
saxpy.uneq	2.84
G.M.	1.90

Table 6. Bound on superscalar instruction issue assuming that the memory (or cache) supplies a maximum of one word per cycle

The peak instruction issue of a superscalar processor is limited to $\frac{BW_s}{f_m}$ instructions per cycle where BW_s is the bandwidth that the data memory can supply and f_m is the fraction of all instructions that are loads and stores [20]. The large percentage of loads and stores in the instruction stream points to the fact that if the peak bandwidth supplied by the memory is limited to one, a future superscalar processor is not capable of issuing more than 2 or 3 instructions. Table 6 illustrates the bound on instruction issue for each program assuming peak memory bandwidth of one. One might argue that caches will capture the locality and reduce the bandwidth requirement of the processor. Even assuming 100% hit ratio, caches have to provide to CPU this large bandwidth. Sohi and Franklin [20] introduced the notion of cache bandwidth. We emphasize the significance of such a metric and point out the fact that the dynamic instruction mix of modern pipelined architectures suggest that cache bandwidth itself can become a bottleneck in the performance of future high performance architectures. It is essential that future processors use multiported caches [25] or implement cache interleaving or other techniques to increase the bandwidth of caches to more than one element per cycle.

3.2 The Access/Execute Balance

Callahan et al. [3] derived their balance metrics based on the objective of overlapping accessing with floating point operations. Decoupled access/execute (DAE) architectures [22] [26] [11] accomplish exactly this. It is true that *pixie* and *pixstats* simulate execution of the program, and give information about pipeline interlocks, but execution in a DAE paradigm with actual memory latencies and cache configurations would provide a more realistic picture of the program balance. Also, the balance metrics β and β_{mod} section 2.1 do not consider branch instructions or address arithmetic. In this subsection, we study the balance in a DAE architecture and illustrate the imbalance taking into aspect the data cache, and address arithmetic and branch instructions. We also present a multiple queue decoupled architecture that reduces the imbalance.

We first simulate the execution of a DAE machine with one access processor (AP) and one execute processor (EP) as in [11] [12]. The access and execute unit are assumed to have the instruction sets and operation latencies of the MIPS R3000/3010 processors. There is a load queue and store queue in the architecture, and they can be specified as sources and destinations (as appropriate) in the instructions. Appropriate modifications in the instruction set for handling the queues is assumed. For array accesses with addresses known at compile-time, the access processor manages the loops and sends an end-of-data token through the queues, after fetching the entire data structure. The execute processor performs the computations until it receives the end-of-data token. Appendix III shows a sample loop coded for the DAE architecture. The number of instructions executed by the access processor and execute processor for single precision data are presented in Table 7.

Benchmark	AP instr count	EP instr count	AP count/ EP count
convolution	5000	3975	1.26
correlation	11083	5926	1.87
saxpy.unequal	12253	2251	5.44
saxpy.equal	3502	2251	1.56
lll11	2765	2501	1.11
strcpy	10012	6001	1.67

Table 7. AP and EP instruction counts and their ratio

Next, we calculate a ratio of the execution times of the AP and the EP. We define AP stand-alone time as the time taken by the AP to complete the access part of the program assuming a perfect EP. Similarly, EP stand-alone time is defined as the time taken by the EP to complete its share of the code assuming a perfect AP and perfect memory. We define AP/EP skew as the ratio of AP stand-alone time to EP stand-alone time. Previous research [11] [12] illustrates that data caches are useful in DAE architectures. The DAE architecture that we simulated uses a 1 kbyte data cache and the AP stand-alone time shown does include the effect of the data cache. We actually performed simulations without the cache also. In the programs that we executed on the DAE simulator, the data cache was seen to improve the performance of only convolution, correlation and strcpy benchmarks. (The cache is 8-way set associative and uses 8 byte blocks. The cache size is kept small so that the entire data set does not fit into the cache.) In Table 8 we present AP/EP skew for memory cycle time t equal to 5, 10, and 15 cycles. (A vlaue greater than 1 indicates AP bottleneck and a value less than 1 indicates EP bottleneck.) The values in parenthesis shows the ratio without the cache. In the benchmark column, .S represents single precision floating point data, and .D indicates double precision floating point data.

Benchmark	t=5	t=10	t=15
conv.S	0.65 (1.42)	0.75 (2.54)	0.84 (3.75)
conv.D	1.55 (2.12)	2.33 (4.00)	3.12 (5.95)
corr.S	1.10 (1.42)	1.13 (2.29)	1.18 (3.39)
corr.D	1.03 (1.89)	1.47 (3.39)	1.17 (5.07)
saxpy.un.S	3.43 (3.43)	5.24 (5.24)	7.83 (7.83)
saxpy.un.D	3.99 (3.99)	6.86 (6.86)	10.30 (10.30)
saxpy.eq.S	2.62 (2.62)	5.20 (5.20)	7.79 (7.79)
saxpy.eq.D	3.33 (3.33)	6.65 (6.65)	9.96 (9.96)
lll1.S	0.90 (0.90)	1.80 (1.80)	2.69 (2.69)
lll1.D	1.33 (1.33)	2.65 (2.65)	3.97 (3.97)
lll3.S	1.22 (1.22)	2.43 (2.43)	3.64 (3.64)
lll3.D	1.79 (1.79)	3.56 (3.56)	5.34 (5.34)
lll11.S	3.13 (3.13)	6.20 (6.20)	9.28 (9.28)
lll11.D	3.85 (3.85)	7.66 (7.66)	11.47 (11.47)
strcpy	1.57 (3.34)	1.89 (6.65)	2.20 (9.98)
G.M.	1.80(2.15)	2.98(4.05)	4.00(6.04)

Table 8. AP/EP skew with (and without) data cache

The skew indicated in Table 8 is analogous to program balance in section 2.1. Actually β (in section 2) gives preliminary indications about program imbalance and the skew presents more accurate information. The direction of the skew in Table 8 indicates that immediate attention is required in speeding up the AP and the memory relative to the EP performance. It may be observed that this was implied by the instruction mix and β in section 2 also.

The skew presented in Table 8 is the combined effect of instruction imbalance and insufficient memory bandwidth. We isolate the memory bandwidth by computing the memory access time for each program. The ratio of time spent in memory access to EP stand-alone time before and after introducing the data cache is illustrated in Table 9. The results indicate that bandwidth is a bottleneck in many programs even after the use of a data cache. As mentioned in section 2.5, several compiler optimization techniques are reducing the temporal locality in programs and this contributes to loss of effectiveness of the data cache in many programs.

Program imbalance as portrayed in section 2 and the AP/EP imbalance in DAEs illustrated in this sub-section point to the fact that DAE architectures should be designed to handle an access process that is heavier than the execute process. The simplest solution to this is to incorporate more load/store units in the access processor. We propose a decoupled architecture with multiple load/store units and multiple queues as in Fig. 3. The system is comprised of an execute processor and an access processor. The access processor is comprised of two address generation and load units (AGLUs) and one address generation and store unit (AGSU). Each AGLU and AGSU has a control queue (CQ1, CQ2 and CQ3 in Fig. 3) through which the EP can request AGLU/AGSU action. The AGLUs and AGSU perform indexing and other address calculation instructions and can issue memory reference instructions. Two register addresses, say 30 and 31 are allocated for the two load queues, and the store queue will be denoted by the register address 31 because load queues can only be read and store queue can only be written from the EP side. The two-load-queue structure will be very useful in programs that perform computations on two arrays simultaneously. The two queue structure can be utilized even in programs loading only one data structure. Alternate elements can be loaded to alternate queues, and the EP program loop can be unrolled an odd number of times so that R31 and R30 are specified as sources in the alternate iterations. A multi-ported data memory is essential for the successful operation of decoupled systems with multiple load/store units.

Benchmark	t=5 cycles		t=15 cycles	
	without	with	without	with
avg of 3	1.63	0.27	4.88	0.8
avg of 8	1.81	1.33	5.43	3.67

(avg of 3 is the average of conv, corr, and strcpy)
(avg of 8 is the average of all 8 programs)

Table 9. Ratio of time spent in memory access to EP stand alone time (with and without data cache)

To illustrate the operation of the multiple load and store units, a program example is provided in Appendix IV. The code segments for the AGLUs and the AGSU are written in elementary operations in a MIPS style. The AGLUs and the AGSU can be designed to include autoincrementing or update instructions or branch with decrement instructions. The Wm architecture [26] has four data fetch units and four data store units and bears some similarities to our multiple queue decoupled architecture. If we were to provide several load and store units, we would have provided one store unit for every two load units, because the frequency of store instructions is approximately half of that of load instructions.

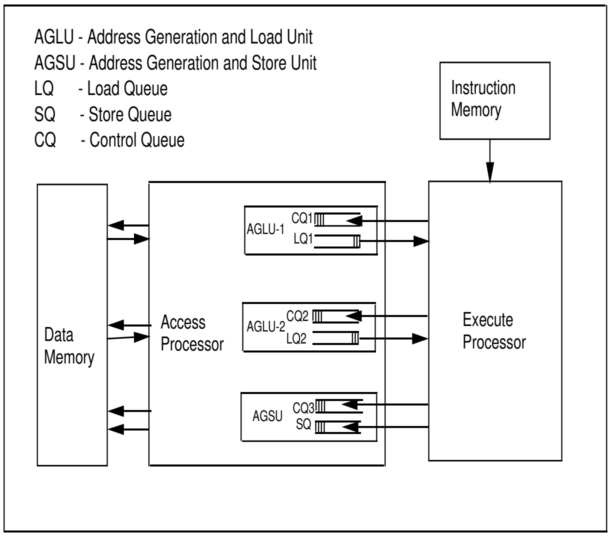

Figure 3: A decoupled architecture with multiple load/store units and multiple queues

We simulated the multiple queue decoupled architecture and studied the imbalance between access and execute processes under that condition. The ratio of the instruction counts in a decoupled architecture with one load/store unit and the architecture with three load/store units is shown in Table 10. There are 3 threads of operation in the AP. By AP instruction count, the number of instructions (or basic operations) in the longest thread in the AP is referred to. Except in *saxpy.unequal* where the compiler could not remove the address arithmetic in each iteration, the access and computation workloads are almost matched.

It may be beneficial to incorporate one load/store unit for each data structure being accessed. In computations of the form V = V + V*V, where V denotes a vector, efficient operation would need 3 load units and 1 store unit, especially if an add-multiply pipeline (as in the IBM RS/6000) exists in the EP. But it is yet to be seen how frequently such computations arise, and how utilized the third load unit would be.

Benchmark	Single L/S unit	3 L/S units
convolution	1.26	0.72
correlation	1.87	1.33
saxpy.unequal	5.44	2.77
saxpy.equal	1.56	1.11
strcpy	1.67	1.00
lll11	1.11	0.71
G.M.	1.83	1.13

Table 10. Ratio of AP and EP instruction counts (with single load/store unit and with 3 load/store units)

3.3 Impact of Wider Bus and Higher Memory Bandwidth

The multiple queue decoupled architecture solves the imbalance between access and execute processes, but it may be very expensive to construct a memory with 3 ports and 3 buses. We thought that it is worthwhile to investigate the effect of doubling the bus width on a uniprocessor. In order to obtain preliminary indications on the balance of a n-bit architecture with 2n-bit data bus, we decided to assume an ISA similar to that of R3000/3010, but with 64-bit data paths to memory. The immediate effect of the assumption would be that the number of floating point loads get halved. We recomputed program balance under the new assumption of double bus-width and double memory (cache) bandwidth and the results are presented in Table 11.a and Table 11.b.

benchmark	%Load + %Store	%fl. pt	β	β_{mod}
tomcatv	38.8	44.7	0.87	0.69
dnasa7	31.5	29.1	1.09	0.67
fpppp	43.5	34.7	1.25	0.81
doduc	21.9	27.9	0.78	0.31
matrix300	38.3	25.5	1.51	1.34
G.M.	33.9	31.7	1.07	0.69

Table 11.a Balance in the SPEC floating point programs on the MIPS assuming double bus-width and double memory bandwidth

Benchmark	%Load + %Store	%fl. pt	β	β_{mod}
concat.c	38.12	41.57	0.92	0.54
concat.f	50.94	38.79	1.31	1.03
conv	49.14	24.16	2.03	1.60
corr	25.78	26.28	0.98	0.97
saxpy.eq	52.79	34.68	1.52	1.22
saxpy.uneq	21.43	13.96	1.54	1.54
G.M.	37.50	28.13	1.33	1.08

Table 11.b Balance in the small benchmark programs assuming double bus-width and double memory bandwidth

The balance metric reduces to 1.33 and 1.07 for the small programs and SPEC benchmarks ignoring pipeline interlocks. Considering the interlocks as well, the respective numbers are 1.08 and 0.69. Thus we observe that doubling bus and memory bandwidth is required to establish balance between accessing and computations in the MIPS R3000/3010 system. It may be remembered that we did not change our assumptions about the floating point unit. Any improvements within the floating point unit will aggravate the balance unless accompanied by similar improvements on load/store issue, bus width and memory bandwidth.

3.4 Further Comments

Number and nature of functional units in other fine-grain parallel architectures:

Extensive research has been done on the mix of functional units required for a fine-grain parallel processor [2] [21] [9]. The significance of providing a close match between the number and nature of functional units in a system and the dynamic instruction mix of the applications running on the system was emphasized by Butler et al. [2]. Quoting Butler et al., "Machine resources must reflect the instruction frequency if we are to achieve efficient utilization of functional units. ...... If integer operations comprise nearly 38% of the instructions in the integer benchmarks and the machine contains only one integer unit, then it is unreasonable to expect a speed up of much greater than two, simply because the integer ALU will be saturated." Along the same lines, we put our argument that if two-thirds of all instructions are access related and if only one access instruction is issued every cycle, one would not obtain a speed up of much more than 1.5. There will be endless combinations and permutations of functional units that one could experiment with. Program characteristics vary widely between applications and it is difficult and perhaps impossible to find a 'magic configuration', ideal for all applications. Nevertheless, our studies emphasize the requirement to provide sufficient number of load/store pipes and the need for a large bandwidth from the main memory and caches.

Impact on Prefetching/Latency Hiding Techniques:

In the recent past, as the speed disparity between processors and memory has grown, there has been fervent research on hardware and software cache prefetching and other latency hiding techniques. The behavior of the instruction stream as presented in section 2 can have a serious impact on many of those techniques.

Consider the simplest and time-tested mechanism of caches. Multi-word caches capture spatial locality and reduce effective access time. But as larger block sizes to capture more spatial locality are employed, there is the risk that all elements fetched may not be used by the program, and that results in further increasing the bandwidth requirements of the program. This is particularly true in the case of nonunit strides. We observed in section 2 that bandwidth is a bottleneck in most supercomputing applications. Hence caches with large block sizes may constitute a burden in such programs if all elements in the cache block are not used. This would not have been the case if extra bandwidth was available.

Now consider software cache prefetching [19] [6] [10] [16]. In the simplest mechanism to insert advisory prefetch instructions, a prefetch is added for every load instruction. If loads constitute almost 40% of all instructions in the instruction stream, indiscriminate software prefetching results in increasing the code size by 40%. The need to issue so many extra instructions can nullify the benefits of prefetching itself. This problem has been solved by previous researchers by generating prefetches only for potential cache misses [19] [10] [16]. If the instruction stream contained a fairly low percentage of loads, there would have been no requirement for such optimizations. Another issue is that indiscriminate (unoptimized) software prefetching may increase the memory bandwidth requirement. If computer systems had "lots and lots of extra bandwidth" [17], this would not have been a problem, but in the light of the scenario in section 2, it is a serious problem. In software prefetching, if the address generated for prefetching cannot be preserved until the actual load, duplicate address generation may be required which can increase the code size. Our studies show that the SPEC floating point programs generate roughly 13% address computation instructions, which indicate the worst case overhead that may arise due to duplicate address generation.

All anticipatory cache prefetching policies can result in superfluous fetches and extra memory traffic. In the case of control dependencies, incorrect branch predictions or aggressive prefetching along both paths create superfluous fetches and extra memory traffic. Since memory bandwidth is already a bottleneck, aggressive prefetching can reduce performance rather than improve it.

The bottom line, as Mowry et al. [16] also pointed out, is that latency can be hidden only if extra memory bandwidth is available. Our studies on program balance suggest that techniques such as blocking [13] that optimize locality and reduce bandwidth requirements of the program are more relevant than latency hiding techniques.

Caches capture locality and reduce memory bandwidth requirements of programs. But as pointed out in section 2.5, common subexpression elimination and avoiding RAW (Read after Write) hazards by using registers and many other common optimizations of code reduce temporal locality in code, and that reduces the impact of caches. As mentioned in section 3.2, out of the 8 traces we used for simulations, only 3 benefitted from data caches.

4 Conclusions

In this paper, we studied the dynamic instruction stream of the MIPS processor while executing the SPEC floating point programs and several floating point kernels and observed that these programs place heavier demands on the memory system than on the floating point units in the R3000/3010 system. On the average, every other instruction in the dynamic instruction stream of these programs is a load or store instruction. Including address computation, access related instructions constitute roughly two-thirds of all instructions executed by typical floating point programs. The imbalance in the instruction stream has a major impact on the design of future processor and memory systems. If two-thirds of all instructions are access-related and if only one access instruction is issued every cycle, one would not obtain a speed up of much more than 1.5. The computing power in many of today's computers exceeds their accessing power and computer designers and architects should emphasize on balancing this mismatch. We presented the design of a decoupled access/execute architecture that can issue up to three access instructions in a cycle. With the multiple load/store units that it has, it is successful in balancing the access process with the execute process.

The instruction stream statistics presented in the paper point to a big demand from cache and memory in terms of bandwidth. Aggressive instruction issue is being studied by many researchers. Unless supplemented by equivalent memory systems, Amdahl's law shows that the performance of computer systems will be limited by memory bandwidth. Single instruction stream parallelism will not be much greater than two, if memory bandwidth is limited to one. The Cray supercomputers always provided plenty of extra memory bandwidth [17]. As large amounts of high performance computing is getting shifted to smaller computers, their designers should be aware of the demands made by supercomputing applications on the memory system.

References

[1] Becker J.C. and Park A., "An Analysis of the Information Content of Address and Data Reference Streams", Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems, May 1993, pp. 262-163.

[2] Butler M., Yeh T.Y., Patt Y., Alsup M., Scales H., and Shebanow M., "Single Instruction Stream Parallelism is Greater Than Two", Proc. of the 18th Annual International Symposium on Computer Architecture, Toronto, Canada, May 1991, pp.276-286

[3] Callahan D., Cocke J., and Kennedy K., "Estimating Interlock and Improving Balance for Pipelined Architectures", Proc. Intl. Conf. on Parallel Processing, 1987, Vol I., pp. 295-304.

[4] Cmelik Robert F., Kong Shing I., Ditzel David R., and Kelly Edmund J., "An Analysis of MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks", Proc. 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, 1991, pp.290-302.

[5] Z. Cvetanovic and D. Bhandarkar, "Characterization of Alpha AXP Performance using TP and SPEC Workloads", Proceedings of the 22nd International Symposium on Computer Architecture, 1994, pp. 60-70.

[6] Gornish E., Granston E., and Veidenbaum A., "Compiler directed Data Prefetching in Multiprocessors with Memory Hierarchies", 1990 International Conference on Supercomputing, pp. 354-368.

[7] Hall Brian C. and O'Brien Kevin, "Performance Comparison of Architectural Features of the IBM RISC System/6000", Proc. 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, 1991, pp. 303-309.

[8] Hammerstrom D.W. and Davidson E.S., "Information content of CPU memory referencing behavior", 4th Annual Symposium on Computer Architecture, March 1977, pp 184-192.

[9] Jouppi N.P. and Wall D., "Available Instruction Level Parallelism for Superscalar and Superpipelined machines", 3rd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems, April 1989, pp.272- -282.

[10] Klaiber A.C., and Levy H.M., "An architecture for software-controlled data prefetching", 18th Intl. Symp. on Computer Architecture, May 1991, pp. 43–53.

[11] Kurian L., Hulina P.T., and Coraor L.D., "Memory Latency Effects in Decoupled Architectures with Noninterleaved Memory", Proc. 19th Intl. Symposium on Computer Architecture, Australia, May 1992, pp. 236–245.

[12] Kurian L., Hulina P.T., and Coraor L.D., "Memory Latency Effects in Decoupled Architectures", IEEE

Transactions on Computers, Vol. 43, No. 10, October 1994, pp. 1129 - 1139.

[13] M.S. Lam, E.E. Rothberg and M. E. Wolf, “The Cache Performance and Optimizations of Blocked Algorithms”, Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 1991, pp. 63-74.

[14] Melvin S. and Patt Y., “Exploiting Fine–Grained Parallelism Through a Combination of Hardware and Software Techniques”, Proc. of the 18th Annual International Symposium on Computer Architecture, Toronto, Canada, May 1991, pp.287-296

[15] S. Mirapuri, M. Woodacre, N. Vasseghi, “The Mips R4000 Processor”, IEEE Micro, April 1992, pp. 10-22.

[16] Mowry T.C., Lam M.S., and Anoop Gupta, “Design and Evaluation of a Compiler Algorithm for Prefetching”, Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems”, October 1992, pp. 62 – 73.

[17] National Science Foundation Memory Workshop Report, April 1993.

[18] Pleszkun A.R., Sohi G.S, Kahalleh B.Z., and Davidson E.S., “Features of the Structured Memory Access (SMA) Architecture”, Third IEEE Computer Society International Conference, San Francisco, CA, March 1986.

[19] Porterfield A. K., “Software Methods for Improvement of Cache Performance on Supercomputer Applications”, Ph. D. dissertation, RICE COMP TR 89-93, May 1989.

[20] G. S. Sohi and M. Franklin, “High Bandwidth Data Memory Systems for Superscalar Processors”, Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, April 1991, pp. 53-62.

[21] Smith M.D, Johnson M. and Horowitz M.A, “ Limits on Multiple Instruction Issue”, Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems, 1989, pp. 290–302.

[22] Smith J.E., Weiss S. and Pang N.Y, “A Simulation Study of Decoupled Architecture Computers”, IEEE Transactions on Computers, Vol.C-35, No.8, August 1986.

[23] Smith W.M, Abraham S.G. and Davidson E.S, “A performance comparison of the IBM RS/6000 and the Astronautics ZS-1”, IEEE Computer, January 1991.

[24] Stephens C, Cogswell B, Heinlein J, Palmer G and Shen J, “Instruction Level Profiling and Evaluation of the IBM RS/6000”, Proceedings of the International Symposium on Computer Architecture, Toronto, Canada, May 1991, pp. 180-189.

[25] Wolfe A and Boleyn R, “Two-ported Cache Alternatives for Superscalar Processors”, Proceedings of the 26th Annual International Symposium on Microarchitecture, December, 1993, pp. 41-48.

[26] Wulf W.A., “Evaluation of the Wm Architecture”, Proc. 19th Intl. Symp. on Computer Architecture, Australia, May 1992, pp.382–390

Appendix I

This appendix contains an example of how our profiling program would tag the instructions to access, execute, address arithmetic, overhead memory reference or branch related types. This is the inner loop for saxpy with equal increments, when compiled with the -O1 option. The less optimized version is chosen for illustration so that the loop contains a few overhead memory references. Most of the overhead memory accesses are eliminated by compiler in the -O4 option.

(a) Source - saxpy with equal increments

```
   do 10 i=1,n
   y(i)=y(i)+a*x(i);
10 continue
```

(b) tagged code

```
$32: lw   $24,8004($sp)  ; load current i to R24     ;(OMA)
                         ; Overhead memory access
     mul  $25,$24,4      ; i times data size to R25  ;(ADDR)
                         ; Address arithmetic
     addu $8,$sp,8008    ; Array start address       ;(ADDR)
                         ; Address arithmetic
     addu $9,$25,$8      ; Base address in R9        ;(ADDR)
                         ; Address arithmetic
     lw   $10,-4008($9)  ; operand x(i) to R10       ;(ACC)
                         ; Data access
     mul  $11,$15,$10    ; a*x(i) – R15 contains a   ;(EXE)
                         ; Execute
     lw   $12,-8008($9)  ; operand y to R12          ;(ACC)
                         ; Data access
     addu $13,$12,$11    ; y(i) + a*x(i)             ;(EXE)
                         ; Execute
     sw   $13,-8008($9)  ; save new y                ;(ACC)
                         ; Data access
     lw   $14,8004($sp)  ; load current index        ;(OMA)
                         ; Overhead memory Access
     addu $24,$14,1      ; increment index           ;(ADDR)
                         ; Address arithmetic
     sw   $24,8004($sp)  ; store new index back      ;(OMA)
                         ; Overhead memory Access
     blt  $24,1000,$32   ; inner loop ends           ;(BR)
                         ; Branch
```