Accounting for the performance of Standard ML on the DEC Alpha

George C. Necula
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Lal George
Rm.2A426 AT&T Bell Labs
600 Mountain Ave.
Murray Hill, NJ 07974

September 20, 1994

Abstract

The performance of several Standard ML (SML) programs on the DEC Alpha 3000/600 is accounted for using the built-in hardware performance counters. The counters provide detailed information of the processor state during execution such as: total instructions, multiple-issue, stalls, cache behavior, and classification of instructions executed. The purpose of this paper is to determine how well the current Standard ML of New Jersey (SML/NJ) compiler utlilizes the DEC Alpha processor. Surprisingly, the processor is stalled for 60-70% of the total cycles executed for most benchmarks — resource conflicts and data cache misses accounting for a large fraction of these. The paper also provides the experimental data needed for an informed choice of which future optimizations to implement.

1 Introduction

Modern superscalar/superpipelined processors like the DEC Alpha 21064 or the Intel Pentium pose a major challenge to compiler writers. Back end optimizations emphasize better usage of processor pipelines and superscalar capabilities over generating the fewest number of instructions possible. Also the usage patterns of the memory subsystem have a significant impact on performance, and optimizations like prefetching, [1, 6] speculative execution,[2] or maintaining data alignment are crucial.

The motivation for this work was to give compiler writers an insight into the processor level behavior of Standard ML (SML) programs on Alpha machines. We provide detailed experimental information about pipeline and cache events during execution. Without this kind of information the process of fine tuning the compiler by implementing new optimizations is merely based on guesswork. For complex architectures like the DEC Alpha, optimizations chosen without precise experimental data are unlikely to perform as anticipated.

The DEC 21064-AA[4] microprocessor provides a set of non-intrusive hardware counters that can be used to measure a variety of execution characteristics. In the past, similar execution profiles were obtained from detailed pipeline simulations — a process that was both tedious and time consuming.

All the SML programs we used were compiled with Standard ML of New Jersey v1.05 compiler (SML/NJ)[**?**].

2 DEC 3000/300 AXP Description

All experiments were run on a DEC 3000/600 AXP workstation equipped with a 175 MHz DECchip^{TM} 21064-AA microprocessor,[4] 64 Mbyte of main memory, and 2 Mbyte of secondary cache.

The DECchip^{TM} 21064-AA is the first implementation of the Alpha architecture. It is a 64-bit RISC processor with separate pipelines for integer, floating point and memory operations. The integer and memory pipelines are seven stages long and the floating point pipeline is ten stages long. The first four stages of all pipelines are shared and implement instruction fetch, branch prediction, resource availability checks, and multiple instruction issue. Upto two instructions can be issued every cycle.

Of particular importance for SML/NJ programs is the behavior of the memory pipeline related to non-blocking loads. The execution will proceed past a load that misses the cache up to a memory instruction or any instruction that uses or defines the register to be loaded. As an exception two other loads that hit the cache are permitted to finish but only if they immediately succeed the load that missed the cache.

The DECchip^{TM} 21064-AA will only dual issue if the two instructions are both in the same aligned 64-bit word and if both have all the necessary resources to complete. Instruction classes that can dual issue are arithmetic (integer or floating point) with memory instructions, and integer arithmetic with floating point arithmetic instructions.

The memory hierarchy on this machine consists of a split instruction/data (`Icache/Dcache`) first level cache built in the processor, a 2 Mbyte second level external cache, and main memory.

The first level caches are: 8 Kbyte each, direct mapped, and have 32 byte cache lines. The write policy on these caches is write-around. This means that a first level write miss will be sent directly to the next cache level, leaving the old contents of the cache in place. The 2 Mbyte second level cache is direct mapped, unified instruction and data, and uses a write-back write policy. We measured the read latency of the secondary cache (a first level cache miss and a second level cache hit) to be 10 cycles. The read latency of the 64 Mbyte main memory, determined experimentally, is 48 cycles. Table 1 summarizes these numbers.

Clock Frequency	150 MHz
On-chip Cache	direct mapped
- size	8 Kbyte data
	8 Kbyte instruction
	32 byte lines
- latency	3 cycle (Figure 2.5, page 2-21[4])
- write policy	write-through, write-around
Secondary Cache	direct mapped
- size	2 Mbyte
- latency	10 cycles
- write policy	write-back, write-around
Memory	64 Mbyte
- latency	48 cycles

Table 1: DEC 3000/600 AXP

Diwan, et.al.,[5] show that the write-no-allocate miss policy is not well suited for the SML/NJ programs, and (correctly) predict poor memory performance on the Alpha. SML/NJ is a memory intensive system with an allocation rate as high as 15 Mbytes/sec. (Version 0.93). A large percentage of objects allocated are short lived, which means that they will be discarded soon after their creation. The allocation of objects will miss in the first level cache as the write occurs to a fresh, uninitialized portion of the heap. Due to the write-no-allocate write-miss policy, the first level cache is bypassed, and the write is sent to the next level. Without prefetching, the subsequent use of the object will again miss in the first level cache. The net effect is poor utilization of the caches.

The DECchip^{TM} 21064-AA is equipped with a write buffer that acts as a high-bandwidth receiver for stores. It has four entries each 32 bytes long. The write buffer will accept a write every cycle and will steal memory cycles to write the data to the secondary cache. Another purpose of the write-buffer is

to coalesce the writes to the same or consecutive memory addresses, resulting in fewer external write transactions. The latter function of the write-buffer is very important for heap based compilers because the vast majority of the writes will be to the top of the heap and therefore adjacent.

The processor incorporates branch prediction logic that attempts to minimize the amount of unused cycles due to a change in the instruction stream. For every conditional branch the processor will predict the branch as taken if it is a backward jump and not taken otherwise. The branch prediction logic will then inform the prefetcher of the new fetching address and will also record in the instruction cache entry of the branch instruction a history bit indicating the final outcome of the conditional branch. If the branch instruction survives in the cache, the next time it is predicted the history bit will be used.

Indirect jumps must be predicted at compile time by encoding the least significant 14 bits of the most probable target address in the instruction word itself. To enhance the prediction of subroutine returns the processor maintains a four-entry return address stack.

Any change in the instruction stream, predicted or not, incurs a penalty. The penalty is one cycle if the conditional branch was correctly predicted, 4 cycles if it was misspredicted and 5 cycles if the change is due to an indirect jump. This is because the branch prediction happens in the second stage of the pipeline while the actual target computation occurs in the fourth or fifth stage of the memory pipeline. At that time one or more useless instructions would have been prefetched in earlier stages. If the pipeline is stalled for some reason and it contains useless instructions, called pipeline bubbles, the prefetcher will continue prefetching instructions and will advance the lower end of the pipeline *squashing* out the bubbles.

3 Performance Counters

The DECchip^{TM} 21064-AA contains a pair of hardware performance counters that can be used to count 17 processor events or instruction statistics. Figure 1 shows the 13 events that we are measuring for SML/NJ programs.

The counters cannot be read but instead an interrupt service routine counts the number of overflows. The user program selects the event to be measured on each hardware counter and also the overflow threshold. Possible values for the threshold are 2^8, 2^{12}, and 2^{16}. There are restrictions on which events can be measured simultaneously and which threshold is valid for each counter.

There are some drawbacks to the current implementation of the performance counters. To measure all 13 parameters at least seven runs of the application are required. Further, since interrupts are generated, the interrupt service routine will be invoked at every overflow. This overhead is likely to influence the number of measured pipeline events and also the cache behavior. To reduce the influence of this factor we only measured one event at a time and only with the highest possible overflow threshold (least number of overflows).

On our machine there was no way to count cache events at the level of the secondary cache. Therefore even if we know the total number of `Dcache` misses we do not know how many were serviced from the secondary cache and how many from main memory. Such information would be extremely important for estimating the total effect of memory latency. Furthermore it would have been useful to know what percent of resource contention cycles are due to the floating point pipeline and to the integer pipeline.

To access the performance counters on OSF/1 v2.0 we use a device driver with a `IOCTL` interface. This device driver is not present in the standard kernel therefore a kernel reconfiguration was necessary. Our experiments with small assembly language programs have shown a measurement error of 1-3%. Although the device driver enables the performance counter interrupts only during the execution of the relevant process, we found that doing the experiments in periods of high system usage influence the experimental results.

CODE	Comment
BRANCH	Counts all branches and jumps
BRMISS	Counts both conditional branch and jump misspredictions
CYCLES	Counts total cycles
DCACHE	Counts primary data cache misses
DRY	Counts cycles where nothing is issued due to lack of valid I-stream data. Causes include Icache fill, misspreduction, branch delay slots.
DUAL	Counts cycles of dual issue
FP	Counts total floating point operate instructions that is, no FP branch, load, or store
FROZEN	Counts cycles where nothing is issued due to resource conflicts or Dcache misses
ICACHE	Counts primary instruction cache misses
INT	Counts integer operate instructions
ISSUE	Counts total issues (instructions)
LOAD	Count all Load instructions
STORE	Counts total store instructions

Figure 1: Performance Counter Selections

Name	Lines	Type	Instr ($x10^6$)	Description
mlyacc	7422	INT	132	A parser generator, processing the SML grammar
cml-sieve	1349	INT	698	The prime number sieve program written using CML
knuth-bendix	580	INT	234	The Knuth-Bendix completion algorithm
vboyer	924	INT	96	The Boyer-Moore theorem prover using vectors
ray	869	FP	1365	Ray tracing
simple	904	FP	914	A spherical fluid-dynamics program
nucleic	2309	FP	135	Nucleic acid 3D structure determination
barnes-hut	3019	FP	806	N-body simulation program
mlmix-nbody	-	FP	125	Partially evaluated n-body problem

Figure 2: Benchmark Programs

4 Benchmarks

To measure the performance of SML/NJ programs on the DEC Alpha we used the Standard ML of New Jersey v1.05 compiler and 9 benchmarks. A short description of each benchmark and its size is given in Figure 2. The **Instr** column in Figure 2 gives the number of instructions executed as given by `ISSUE` performance counter.

Five of the benchmarks are floating point, namely: `ray`, `simple`, `nucleic`, `barnes-hut`, and `mlmix-nbody`. `Mlmix-nbody` is program generated and its line count is not meaningful as each word is on a new line.

5 Instruction Statistics

The `LOAD`, `STORE`, `BRANCH`, `INT`, and `FP` performance counters give the total number of issues for loads, stores, branches, integer operate, and floating point operate instructions. Figure 3 shows the breakdown as a percentage of the total number of issues.

The class of instructions we called `OTHER` includes instructions that are not covered by the counters. These include `NOP`; load address instructions using 0 as the base register (used in SML/NJ to load literal values in registers); trap barrier instruction (one is used after every trap generating arithmetic instruction); and PAL instructions (implementation specific instructions used to implement operating system functions like context switching). It is not surprising that the floating-point benchmarks have a larger share of `OTHER` instructions because every floating-point instruction will be followed by a trap barrier.

All benchmarks are *load intensive* with 25% of instructions being loads. Stores account for 15% of all instructions executed and branches are about 12%. The average of just loads, stores and branches across all benchmarks is 52%. The sum of loads and stores is *at least* 15% higher than that reported for C and Fortran benchmarks[3]. This was to be expected because SML/NJ uses boxed representations to compile polymorphic functions.

What did come as a surprise were the floating point benchmarks. `Simple` that was regarded as a floating point intensive program had less than 1% of all instructions executed being floating point. The SML/NJ floating point benchmarks, have a much smaller ratio of floating point instructions than their C counterparts that may have anywhere between 40-50% [3]. This is mainly attributed to the current compiler that does a lot of unnecessary boxing/unboxing of floating point values, and subscript bounds checks associated with arrays.

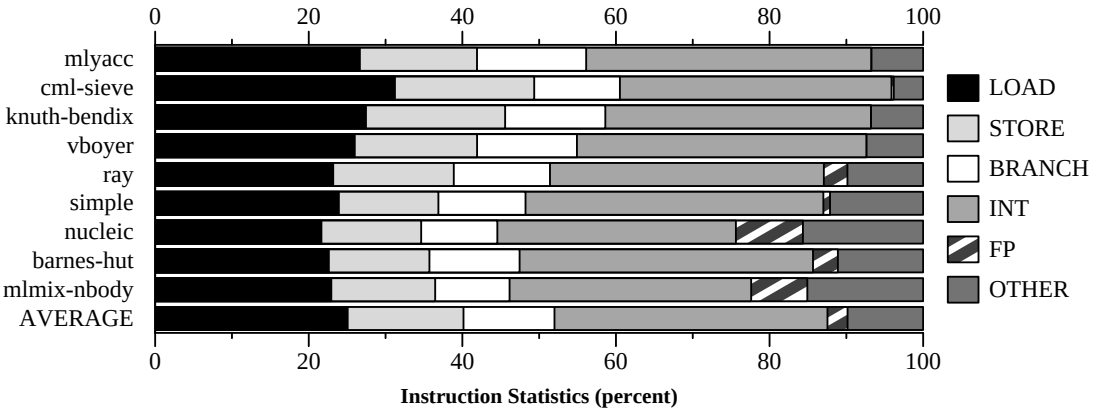

Figure 3: Instruction statistics

6 Cache Misses

Figure 4, summarizes the cache behavior. For all benchmarks the data cache and branch misses are significantly higher than the `Icache` misses.

All `Dcache` misses are the result of load instructions because the store instructions are absorbed by the write-buffer and processed in the background. Figure 4 shows the ratio of `Dcache` misses to the number of load instructions, i.e., (`DCACHE / LOAD`). Approximately 17% of load instructions miss in the primary cache.

The current compiler does very little to optimize branch prediction. Certain infrequently executed blocks such as garbage collection invocation are moved off-line and made into forward branches so that they will not be predicted as taken. Figure 4 shows the ratio of misspredicted branches to total number of branches, i.e., (`BRMISS / BRANCH`). On an average 28% of all executed branches are misspredicted.

Lastly, Figure 4 shows the ratio of instruction cache misses to that of total instructions executed, i.e., (`ICACHE / ISSUE`). On the average 3% of the instructions executed, generated a `Icache` miss. This figure seems extremely low but it must be noted that at every `Icache` miss the processor will bring in the cache 8 adjacent instructions (the cache line is 32 bytes) and therefore an average of 12.5% would mean that almost all instruction cache lines encountered were missing from the cache. Furthermore while a `Dcache` miss does not necessarily stall the processor (because of non-blocking loads), an `Icache` miss will always stop any further issue. Therefore a `Icache` miss could be more expensive than a `Dcache` miss.

7 Pipe Dry

The plots in Figure 4 do not say anything about the impact of cache misses or branch misspredict on performance. The `DRY` performance counter counts the number of cycles the processor was stalled because of the lack of valid instructions due to branches or `Icache` misses. The average percentage of dry cycles was measured to be 22%. It would be interesting to know how many of the dry cycles are due to each of the two possible causes.

We know that in the absence of bubble squashing a predicted conditional branch will cost one cycle and a misspredicted branch 4 cycles. All indirect jumps will have a penalty of 5 cycles as the SML/NJ does not provide any hint to the processor about the target of the jump. The performance counters available in DECchip^{TM}21064-AA give no indication of how many bubbles were squashed. Therefore we cannot precisely state how many `DRY` cycles are due to branches and how many to `Icache` misses.

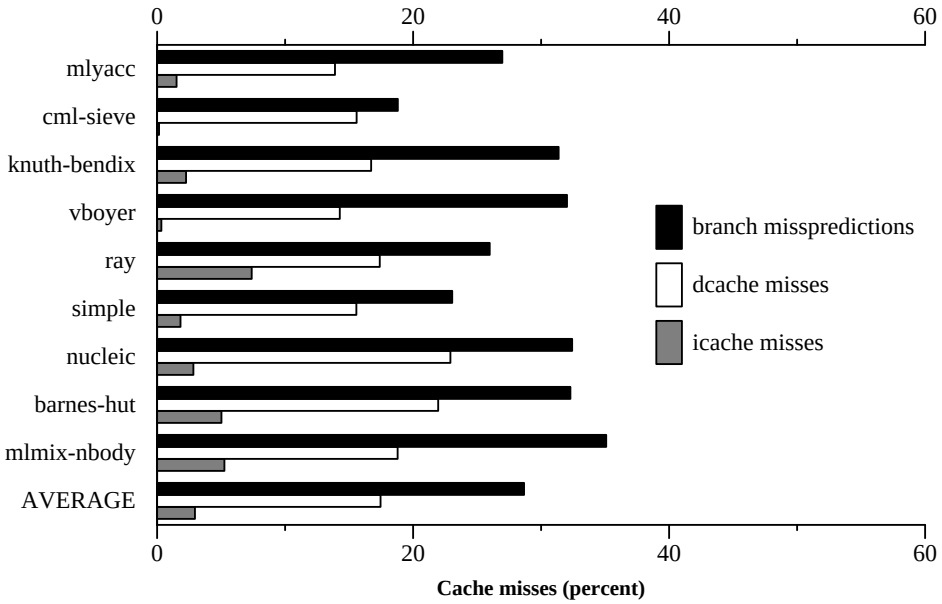

Figure 4: Cache misses

We attempt to estimate the above breakdown by assuming that the proportion of squashed pipeline bubbles equals the proportion of the frozen cycles because during every frozen cycle if there is a bubble in the pipeline it will be squashed. Furthermore we assume that 25% of the branch instructions are indirect jumps. This assumption is backed up by our preliminary measurements on dynamic instruction frequencies using the same compiler. The formula used to estimate the number of dry cycles due to branches is:

$$\frac{((\texttt{BRANCH} - \texttt{BRMISS}) * 1 + \texttt{BRMISS} * 4.25) * \texttt{FROZEN}}{\texttt{CYCLES}}$$

The Figure 5 show the results of our attempt to distinguish among dry pipeline cycles those due to branches and to `Icache` misses. The bulk of the dry cycles is attributed to instruction cache misses even if the percentage of `Icache` misses is small and that of branch misses is large. This is not surprising because the cost of a branch is 1-5 cycles while an `Icache` miss will cost 10-48 cycles.

Branches typically account for 31% of the dry cycles or 6% of all cycles. On the average `Icache` misses account for 69% of the dry cycles or 16% of all cycles.

8 Pipe Frozen

Again, Figure 4 does not say anything about the impact of data cache misses on overall performance. The `FROZEN` performance counter measures the number of cycles the pipeline was stalled as a result of `Dcache` misses or resource conflicts. Our measurements show that these cycles are 41% of all cycles.

It is important for the compiler writer to know how many of the frozen cycles were due to resource conflicts and to `Dcache` misses. We know the number of first level data cache misses given by the `DCACHE` counter. We also measured the penalty of a `Dcache` miss that hits the secondary cache to be 10 cycles. The penalty for a main memory access was measured to be 48 cycles. Again there is no way to precisely compute the number of frozen cycles due to `Dcache` misses because the performance counters on our machine give no information about the secondary cache misses.

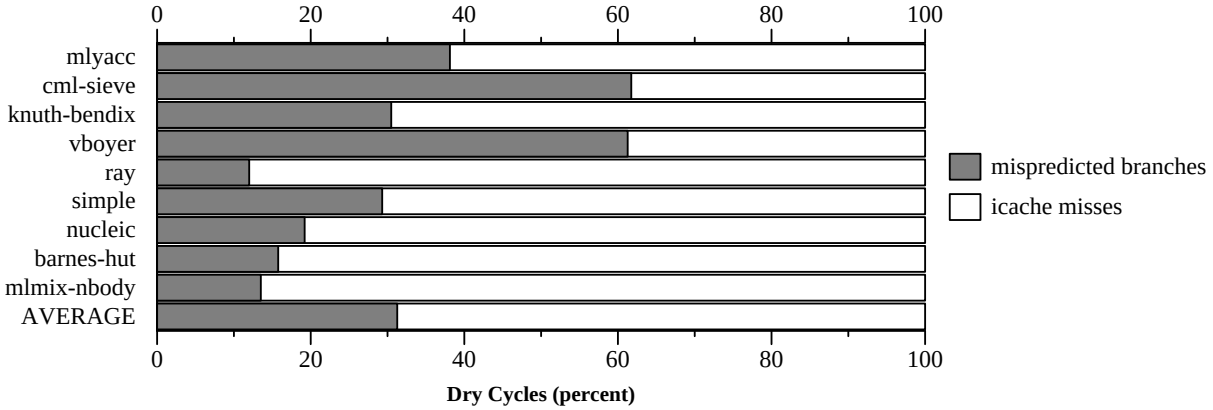

Figure 5: Breakdown of dry cycles between `Icache` misses and branches

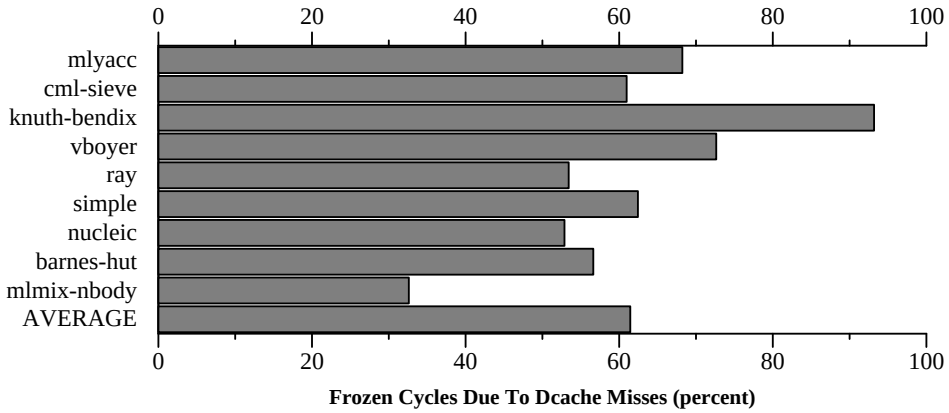

Figure 6: Contribution of data cache misses to frozen cycles

In order to estimate the breakdown of frozen cycles into cycles due to `Dcache` misses and cycles due to resource contention, we made some assumptions. It was shown that for the SML/NJ compiler using a generational garbage collector, about 90% of all loads are from the youngest generation [**?**]. We setup our compiler to use a youngest generation of approximately 80% of the size of the secondary cache. This will insure that the allocation arena will reside in the secondary cache. Therefore we can use the following formula to compute the number of frozen cycles due to memory subsystem latency

$$\texttt{LOAD} * (\frac{90}{100} * 10 + \frac{10}{100} * 48)$$

The bar chart in Figure 6 shows a breakdown of frozen cycles into cycles due to memory latency and frozen cycles due to resource contention.

From Figure 6 the data cache misses must account for 61% of the frozen cycles or 24% of all cycles while resource contention stalls occur during 39% of the frozen cycles or 17% of all cycles.

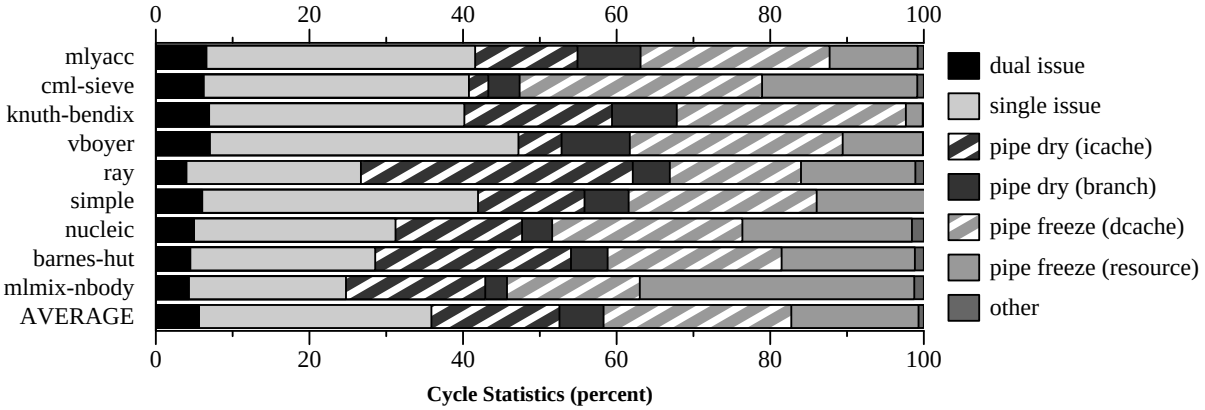

Figure 7: Cycle breakdown

9 Cycle Breakdown

This section summarize the previous sections by giving a cycle breakdown in terms of dual issue, single issue, pipe dry, and pipe frozen cycles. This classification is shown graphically in the Figure 7. As specified in the previous two sections the breakdowns of frozen and dry cycles are not measured experimentally but instead are based on analytical models. In Figure 7 we depict with different shades the percentage of dry and frozen cycles, and among each of these the part due to cache misses is striped.

The number of dual issues is measured directly from the `DUAL` performance counter. The number of single issues (`SINGLE`) is computed using `SINGLE` = (`ISSUE` - 2 * `DUAL`). The cycles classified as "other" are cycles spent in PAL mode and cycles due to the performance counters measuring error.

No attempt is made at satisfying the dual issue constraints by SML/NJ, and the processor does manage to extract a small amount. What is quite startling is that for 60-70% of all cycles, the processor is stalled for one reason or the other, the largest percentage of the stalls coming from frozen cycles.

10 Cycles Per Instruction (CPI)

One of the most widely used indicators of the actual usage of the processor is the CPI (Cycles Per Instruction). A small value of the CPI parameter means a good usage of the processor processing power. The DECchip^{TM}21064-AA implementation of the Alpha architecture is able to execute instructions at a peak value of 0.5 CPI (dual-issue for every pair of instructions). Taking into account the strong dual-issue constraints we feel that a CPI of one is a fair performance to be expected from a compiler.

Using the `ISSUE` and `CYCLES` performance counters, Figure 8 shows the cycles per instruction (CPI) computed using (`CYCLES` / `ISSUE`). The CPI ranges from 1.8 to 3.4 with an average of 2.5. This is only about 25% worse than C benchmarks that have an average between 1.5-2.5 for both integer and floating point benchmarks[3]. The SML/NJ floating point benchmarks have a higher average CPI of 2.9 compared to 2.0 for integer benchmarks.

11 Conclusions

Using the performance counters we have accounted for the performance of SML/NJ programs on the DEC Alpha. For more than 60% of the cycles, the processor is stalled, largely as a result of data cache misses and processor resource conflicts.

The information presented in this paper can be used to select those optimizations that are most likely to improve the performance of SML/NJ programs. For example, our measurements show that an

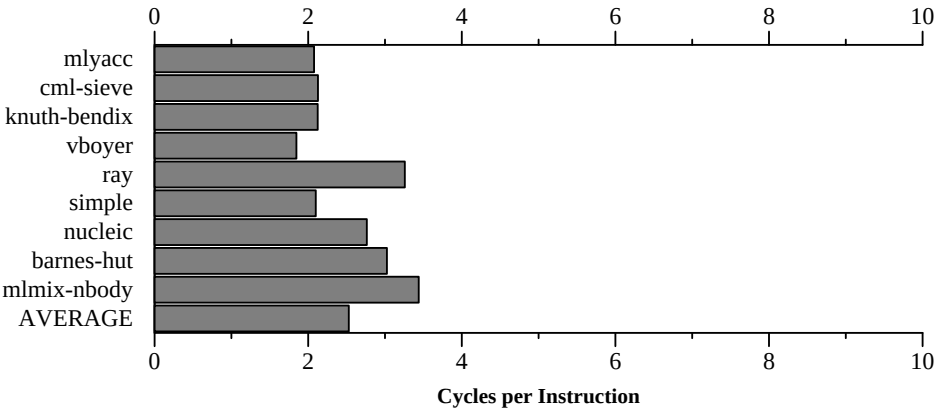

Figure 8: Cycles Per Instruction

optimization aimed at improving branch prediction is going to improve the performance at most by 6%. Implementing code rearrangement to increase the `Icache` hit ratio is likely to give better results because 16% of all cycles are spent waiting for `Icache` misses to complete. But the most profitable optimizations are going to be those which will reduce the number of frozen cycles. A better instruction scheduler is essential because 17% of the time the processor is stalled due to resource contention. The `Dcache` misses account for 24% of all cycles and thus optimizations aimed at reducing the number of these misses will have a large impact on SML/NJ performance.

Considerable work is required in the compilation of floating point programs which tend to exhibit the large CPI values.

References

[1] Appel, A. W. Emulating write-allocate on a no-write-allocate cache. Tech. Rep. CS-TR-459-94, Princeton University, June 15 1994.

[2] Bernstein, D., and Rodeh, M. Global instruction scheduling for superscalar machines. In *Proc. of the ACM SIGPLAN'91 conf. on programming language design and implementation* (June 1991), ACM, pp. 241–255.

[3] Cvetanovic, Z., and Bhandarkar, D. Characterization of Alpha AXP performance using TP and SPEC workloads. In *Proc. of the 21st annual Int. Symp. on Computer Architecture* (April 18 1994), ACM, pp. 60–70. Also Computer Arch. News, Vol 22, No. 2, April 1994.

[4] DEC. *DECchip^{TM}21064-AA Microprocessor*. Digital Electronics Corporation, 1992. Order No: EC-N0079-72.

[5] Diwan, A., Tarditi, D., and Moss, E. Memory subsystem performance of programs using copying garbage collection. In *21st ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (January 1994), ACM, pp. 1–14.

[6] Mowry, T. C., Lam, M. S., and Gupta, A. Design and evaluation of a compiler algorithm for prefetching. In *Fifth intn. conf. on architectural support for programming languages and operating systems* (Sept. 1992), pp. 62–75. Also SIGPLAN Notices vol.27 num.9.