Universidade Federal do Paraná

Departamento de Informática

Fabiany Lamboia & Roberto A Hexsel

An Analisys of Dynamic Instruction Usage with 32 Bit MIPS, PowerPC and SPARC Processors on Embedded Applications

Relatório Técnico
RT_DINF 001/2011

Curitiba, PR
2011

An Analisys of Dynamic Instruction Usage with 32 Bit MIPS, PowerPC and SPARC Processors on Embedded Applications

Fabiany Lamboia & Roberto A Hexsel
Universidade Federal do Paraná
Departamento de Informática
CEP 81.531-990 – Curitiba, PR, Brasil
{fabiany,roberto}@inf.ufpr.br

Abstract

This paper presents a comparison of the instruction set usage of the 32 bits microprocessors MIPS, PowerPC and SPARC. We measured dynamic instruction usage of the 16 programs from MediaBench and 11 from CommBench benchmark suites, all compiled with GCC-3.3.1. The measurements were taken on functional models written in ArchC. The instruction counts for the 27 programs are 24.9, 27.0, and $26.0 \cdot 10^9$ for MIPS, PowerPC and SPARC, respectively. Except for memory references, the number of dynamic instructions for SPARC are larger than the other's, whereas PowerPC has the highest memory reference count. We assessed the effects of three levels of optimization on dynamic instructions and found small differences in instruction counts. We also measured the distribution in time of memory references and control instructions. While running CommBench programs, about 80% of LOADs and 60% of STOREs are separated by less than five instructions. For MIPS and PowerPC, about 80% of conditional branches are separated by up to six instructions, while for SPARC about 90% of branches are separated by up to three instructions.

1 Introduction

This paper presents a comparison of dynamic instruction usage of the 32 bits microprocessors MIPS, PowerPC and SPARC that are popular in embedded applications, typically as part of a larger system (SoC). Sixteen programs from the MediaBench suite [9] and eleven from the CommBench suite [18] were all compiled with the same version of GCC and their execution was simulated on functional models of the CPUs written in ArchC [15]. The simulation results are used to compare the CPUs and code generators, assessing the number of instructions executed, number and distribution of memory references, and number and distribution of flow control instructions.

These studies are not original. There was a comparative analysis of the instruction set architectures (ISA) of popular microprocessors in the first edition of [6] and the current edition compares Alpha, MIPS, PA-RISC, PowerPC, and SPARC [7]. There is also a comparison of MIPS and SPARC ISAs running the SPEC benchmarks compiled with their native compilers and two distinct simulators in [2]. The comparison between Alpha and PowerPC ISAs in [16] provided inspiration for this work. An assessment of SPARC ISA, using the CommBench suite, is presented in [18].

Unlike those, our analysis is conducted "on a level playing field" with the same programs and same simulation framework. Our results can be used (i) to optimize a compiler's

back end, (ii) to design the CPU-memory interface, (iii) to improve the branch prediction implementation, and (iv) ultimately to help in choosing a CPU best suited for a given application.

The text is organized as follows. Section 2 gives a brief description of the three ISAs. The benchmarks and simulation environment are described in Section 3. Our results are presented in Section 4, and in Section 5 we draw our conclusions.

2 Microprocessors

The instruction sets we compare were conceived and designed in the early 1980's and they qualify as *Reduced Instruction Set Computers* or RISCs. One for the main goals of their designers was to implement a complete 32 bit processor in a single integrated circuit and therefore the instructions had to be simple, that is, *reduced*. It is no coincidence the these ISAs are similar because their designs were subject to the same constraints, which can be satisfied by: (i) the operands of logic and arithmetic instructions come from registers; (ii) simple addressing modes, with LOAD instructions reading values from memory and STORE instructions writing results back; (iii) all instructions are the same size and regularly encoded; and (iv) many general purpose registers with the same width.

The descriptions that follow are presented from an abstract point of view that considers an ISA as the hardware's API. In the text we focus on the 32 bit version of the three instruction sets and ignore floating point instructions. To simplify the descriptions, MIPS ISA is presented first and the other two are presented with the same syntax. The instructions are defined with semantics similar to that of C. The description of the ISAs is in historical order; the discussion of results follows an alphabetical order.

2.1 MIPS

The early designs of the *Microprocessor without Interlocked Pipeline Stages* defined the execution model of a five stage pipeline. In this model, an instruction can do just a few simple operations in each pipe stage [5].

The spartan design could not support interlocking to resolve data and control dependencies forcing the designers to expose some of the implementation to the programmer, who had the responsibility of placing an instruction (or a NOP) after a branch instruction to fill the *branch delay slot*. The programmer also had to ensure that an instruction dependent on the result of a LOAD did not use the value in the following cycle, because of the *load delay slot*. Newer versions of the MIPS ISA (MIPS32) have kept the branch delay slot but the implementations usually stall to resolve a load-use hazard. The 32 bit instruction set is defined in [10]. The most relevant characteristics are briefly described below.

Registers 32×32 bit registers. Register r_0 is always zero and stores to it are ignored. Register r_{31} is the link register for function returns.

Logic and arithmetic These instructions take two operands from registers (or an immediate) and place the result in a register. The immediate is a 16 bit value that can be sign/zero extended. Multiplication takes two 32 bit operands and places the 64 bit result in the special registers hi and lo. Division places remainder and quotient in these two registers. Instructions MFHI and MTHI load values from, and into, these registers. The suffix 'U' denotes an instruction that does not check for overflow (unsigned), while signed instructions cause an exception on overflow. The same applies

to the sign extension of immediates: suffix 'U' implies zero-extension, rather than sign-extension. Logic instructions zero-extend the immediate operand.

Memory There is a single addressing mode for memory references: the contents of a base register are added to a 2's complement immediate. Memory is a 4Gbytes vector that can be referenced as 8, 16, and 32 bit aligned bytes, half-words and words.

Conditional branches Branch targets are PC-relative with a reach of ±32K instructions. Conditions are computed into registers as there is no status register for this purpose. There are instructions that do an equality comparison and branch, while magnitude comparisons take two instructions, a compare and a branch.

Unconditional Jumps The addressing mode of jump instructions is called *pseudo-absolute* because the target is specified by juxtaposing the 4 most significant bits from the PC, 26 bits from the operand and 2 zeroes to align the address. Thus, jumps can reach a segment of 64M instructions. The JR instruction uses the 32 bit contents of a register for its target.

Functions The instruction JAL jumps to the target address and saves the next instruction's address in the link register r_{31}. Function return is accomplished with an indirect jump through r_{31}. The target addressing is pseudo-absolute.

Addressing modes There are five addressing modes: (i) *register*– operands come from registers; (ii) *immediate*– one of the operands is an extended 16 bit field from the instruction; (iii) *base-displacement*– the effective address is the sum of a base register with a sign-extended displacement in the immediate field; (iv) *PC relative*– branch target is less than ±32K instructions away from the branch instruction; and (v) *pseudo-absolute*– effective address is obtained by the concatenation of 4 bits from the PC and the word aligned 26 bit operand.

2.2 SPARC

The *Scalable Processor ARChitecture* (SPARC) descends from RISC I [11] and is more similar to MIPS than to PowerPC. The distinguishing feature in this ISA is the organization of the register file as a set of *register windows*, with a new window allocated on each function call. While on a given call depth, the registers are split into four groups, each group used for (i) holding globals, (ii) incoming parameters, (iii) local variables, and (iv) outgoing parameters. When a function is invoked, the window slides and the incoming parameters become the local variables of the called function. The window is used by compilers to create a stack frame directly onto registers thus avoiding (several) memory references. A window has 16 unique registers and implementations support up to eight windows, or 128 registers plus the 8 globals. SPARC instructions are encoded with a 2 bit opcode plus one or two additional fields to define the instruction. Currently `SparcInternational` holds control over the ISA and the 32 bit version is defined in [17].

Logic and arithmetic Instructions can optionally set status bits on the condition code register r_{cc}, and magnitude comparisons are achieved by subtracting from r_0, which is always zero. Immediates are 13 bits wide.

Memory Memory addressing can use indexed addressing: $r_d \leftarrow$ `M[`r_b`+`r_i`]`. The instructions LDD and STD can load or store double words (64 bits) in a single memory reference.

Conditional branches Instructions test the appropriate bit in the condition code register r_{cc}. There is an annulling branch that executes the instruction in the branch

delay slot only of the branch is taken; otherwise that instruction has no effect. The delay slot of ordinary branches must be filled by the programmer.

Functions When all windows are in use, the next function call causes an exception that must save (spill) some registers in memory.

Addressing modes Same as in MIPS ISA, plus indexed. Jump targets are specified with 30 bits, with a 2 bit opcode.

2.3 PowerPC

The *Power Performance Computing* (PowerPC) ISA descends from IBM's 801 [14], and the PowerPC 601 was the first member of the family [1]. The project was developed about a decade later than the other two RISCs described here, and the architecture was conceived for superscalar implementation with three pipelines that operate in a (more or less) decoupled way: a branch unit, an integer unit, and a floating point unit. Because of this partitioning, more complex operations can be performed on each pipeline stage and the designers of the PowerPC added instructions more complex than those of strict RISC processors. The PowerPC architecture is managed by `Power.org` and the most recent version of the ISA is described in [13]. Below we briefly mention the differences with respect to MIPS and SPARC ISAs.

Logic and arithmetic Two operand logic and arithmetic instructions can, optionally, set bits in the condition register r_{cr}. r_0 and r_{31} are general purpose registers.

Memory Data references can use indexed addressing. There are instructions that load or store character strings of arbitrary length and alignment. The instruction LMW (STMW) can load (save) up to 32 registers.

Conditional branches The condition register r_{cr} holds 4 bits that can optionally be set by logic and arithmetic instructions. The branch unit holds eight instances of r_{cr} and thus can hold the status of up to 8 instructions. The branch unit is near the beginning of the fetch pipeline and so branches take effect with little or no delay. There is a dedicated counter register $\mathsf{r}_{\mathsf{ctr}}$ that can hold a loop iteration counter so that a single branch instruction can decrement $\mathsf{r}_{\mathsf{ctr}}$, test against zero and then branch.

Functions Return addresses are kept in the dedicated link register r_{lk}.

Addressing modes There are three addressing modes not found in MIPS ISA: indexed, *base-displacement with increment*, and *indexed with increment*. In these two, the base register is updated with the effective address just computed: $\mathsf{r}_d \leftarrow \mathtt{M[}\langle x = \mathsf{r}_b + \mathsf{r}_i \rangle\mathtt{]}$; $\mathsf{r}_b \leftarrow x$. Jump targets are specified in 24 bits.

3 Simulation Environment

The simulators used to generate the instruction counts were written with the architecture description language ArchC [15] and are available from `http://www.archc.org`. ArchC enhances SystemC [3] so that a simulator can be generated from a description of both the syntax and the semantics of an instruction set. The simulators model the processors as described in [8] (MIPS), [12] (SPARC), and [19] (PowerPC). Notice that these are functional simulators because no timing information is ever computed and each instruction is executed atomically in a simulation cycle that is in no way related to a "clock cycle".

Workload The instruction counts were measured by simulating the execution of 16 programs from the MediaBench suite [9] and 11 programs from the CommBench suite [18]. All programs were compiled with GCC-3.3.1 provided with the simulators, all were optimized with `-O3` and statically linked to libraries that emulate system calls. The SPARC model uses 256 registers, in 16 windows, which is twice the number found in typical implementations. Thus, simulation of function calls with this processor is rather optimistic.

The MediaBench programs used are ADPCM (Adaptive Differential Pulse Code Modulation), EPIC (image data compression), G721 (CCITT G.711, G.721 and G.723 voice compression), GSM (RPE/LTP voice coding at 13 kbit/s), JPEG (image compression and decompression), MPEG (player for MPEG-1 and MPEG-2 video bit streams), and PEGWIT (public key encryption and authentication).

CommBench programs are relatively small and contain kernels typical of the applications run on network processors. The suite contains "header-processing applications" that read and write message headers, and "payload processing applications" that read and modify the payload of messages. The header-processing programs used are RTR (Radix-Tree Routing), FRAG (IP packet fragmentation), and DRR (Deficit Round Robin scheduling), and the payload processing programs were CAST (block ciphering with CAST-128), ZIP (Lempel-Ziv compression), REED (Reed-Solomon encoding), and JPEG (lossy compression of image data).

4 Results

The reader must bear in mind that a comparison of dynamic instruction counts *is not a fair indication of actual performance*. The formula that computes the execution time of a given program on one processor is $time = \mathcal{N} \times CPI \times \mathcal{T}$, where $\mathcal{N}$ is the number of instructions executed, CPI is the average number of cycles per instruction, and $\mathcal{T}$ is the clock period. The ISA has an impact on the number of instructions executed, and its implementation determines both CPI and clock period. Furthermore, the compiler has a strong influence on $\mathcal{N}$ and CPI as it can generate "fast" or "slow" code, as is further discussed in Section 4.2. All else being equal, implementation techniques have a large influence on performance, as for instance, in the implementations of Intel's IA-32 in the 80386 and PentiumIV processors.

Table 1 shows the instruction counts of all programs, and also the totals for CommBench and MediaBench. Instructions are split into classes somewhat arbitrarily and the choices are not trivial: (i) instructions listed as ALU comp could be counted as branches since magnitude comparisons are used to test conditions; (ii) instructions listed as const MS load the upper half of a register and are often used to load an absolute address into a base register and thus could be counted as memory references (which ones?); (iii) not all instructions counted as ret are function returns because the jump-register instruction is also used in jump tables. The histograms in Figure 1 compare the totals as well as the counts of five main instruction classes. The instruction classes are listed below.

ALU logic logic operations: AND,OR,XOR;

ALU arit arithmetic operations: ADD,SUB,MUL,DIV;

ALU desl shift and rotate;

ALU comp magnitude comparisons;

const MS load most significant half of register;

move copy to/from register; MIPS, SPARC: to/from mult/div accumulators;
load LOADs, all sizes;
store STOREs, all sizes;
jump unconditional jumps;
branch conditional branches;
call function calls;
ret function returns and register indirect jumps;
save SPARC: change register window;
restore SPARC: restore register window;
nop MIPS e SPARC: fills delay slot;
total gross total;
ALU total all logic, arithmetic, comparison;
MEM total all memory references;
CTR total all jumps and branches;
FUN total all function calls and returns;
OTH total all other, not ALU, MEM, CTR, FUN.

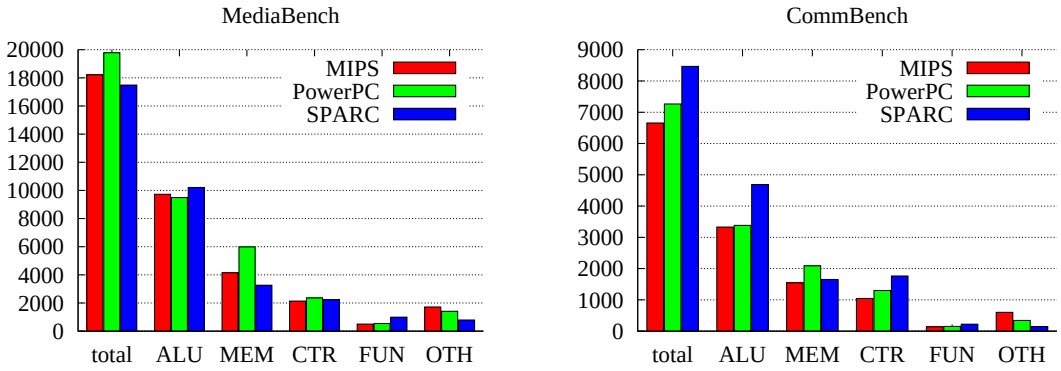

Figure 1: Number of instructions, total and by class (instr.$\times 10^6$)

4.1 Execution profile

Before our first look at the results, we expected that the number of dynamic instructions would ordered as PowerPC, SPARC and MIPS, because of the more complex instructions in PowerPC [16], and of the register windows in SPARC. Contrary to our expectations, the PowerPC executes 10% more instructions than SPARC, 8% more than MIPS, and the culprits seem to be the memory references, as shown in Table 1.

The programs that comprise MediaBench are larger, more general purpose, and process larger data sets than those in CommBench. Therefore we chose to discuss separately the results for the two sets. In the following two sections we focus on the results for CommBench because these programs were selected from a narrower field of application and therefore display behaviors with slightly less entropy.

All	**MIPS**		**PowerPC**		**SPARC**	
class	×1000	%	×1000	%	×1000	%
ALU logic	2,804,816	11.28	3,280,844	12.13	5,390,427	20.77
ALU arit	6,305,592	25.35	3,617,228	13.37	4,881,806	18.81
ALU desl	2,540,121	10.21	3,159,846	11.68	1,834,513	7.07
ALU comp	1,401,805	5.64	2,816,278	10.41	2,787,738	10.74
const MS	380,163	1.53	934,740	3.46	758,031	2.92
move	146,939	0.59	816,883	3.02	71,320	0.27
load	3,446,317	13.86	4,500,029	16.64	3,273,393	12.61
store	2,247,748	9.04	3,573,505	13.21	1,640,929	6.32
jump	290,402	1.17	363,926	1.35	450,186	1.73
branch	2,886,063	11.60	3,294,023	12.18	3,543,752	13.65
call	315,161	1.27	373,507	1.38	334,897	1.29
ret	318,825	1.28	319,395	1.18	339,259	1.31
save	—	—	—	—	271,384	1.05
restore	—	—	—	—	271,384	1.05
nop	1,790,057	7.20	—	—	106,170	0.41
total	24,874,009	—	27,050,203	—	25,955,189	—
ALU total	13,052,334	52.47	12,874,196	47.59	14,894,483	57.39
MEM total	5,694,064	22.89	8,073,534	29.85	4,914,322	18.93
CTR total	3,176,465	12.77	3,657,949	13.52	3,993,939	15.39
FUN total	633,987	2.55	692,902	2.56	1,216,924	4.69
OTH total	2,317,159	9.32	1,751,623	6.48	935,521	3.60
			MediaBench			
total	18,218,559	—	19,787,460	—	17,488,485	—
ALU total	9,723,901	53.37	9,493,691	47.98	10,207,180	58.37
MEM total	4,150,512	22.78	5,984,398	30.24	3,263,276	18.66
CTR total	2,131,439	11.70	2,356,910	11.91	2,230,919	12.76
FUN total	496,579	2.73	543,172	2.75	996,443	5.70
OTH total	1,716,129	9.42	1,409,289	7.12	790,667	4.52
			CommBench			
total	6,655,450	—	7,262,743	—	8,466,704	—
ALU total	3,328,433	50.01	3,380,505	46.55	4,687,303	55.36
MEM total	1,543,552	23.19	2,089,136	28.77	1,651,046	19.50
CTR total	1,045,027	15.70	1,301,039	17.91	1,763,020	20.82
FUN total	137,408	2.06	149,729	2.06	220,481	2.60
OTH total	601,030	9.03	342,334	4.71	144,854	1.71

Table 1: Instruction counts, all programs

MediaBench The graph on the left of Figure 1 shows the results for the 16 MediaBench programs. The totals for the PowerPC are taken as reference and the counts of the other two ISAs are compared to that. MIPS executes 8.6% less instructions than PowerPC, 2.2% more ALU instructions, 44% less memory references, about 10% less flow control and function call/return instructions. MIPS executes 18% more instructions grouped as 'other', that include 1.3×10^9 NOPs, while PowerPC executes three times as many MOVEs and address constant loads. PowerPC's more powerful addressing modes are not being put to good use in these programs.

SPARC executes 12% less instructions than PowerPC, with the memory references accounting for most of the difference. SPARC executes about half (55%) as many memory

references and here the advantage comes from the register windows. For the three processors, the number of function calls and returns is similar (within 20%, less than 1.5% of all instructions) and the advantage in memory references comes from SPARC not saving and restoring activation records. GCC seems to be able to hide 18 times more of SPARC delay slots than those of MIPS (0.07 / 1.26).

CommBench As was done for the MediaBench programs, PowerPC is used as the reference for the comparisons. PowerPC executes 9% more instructions than MIPS, and 17% less than SPARC. Except for memory references, SPARC performs more instructions than the other two. If all the memory related operations are considered (LOADs, STOREs, SAVE, RESTORE, NOP), SPARC performs 19% less memory related instructions than PowerPC and 18% less than MIPS, on the assumption that one half of NOPs are filling load delay slots. If the other half of NOPs is counted as filling branch delay slots, then SPARC performs 36% more control instructions than PowerPC and 34% more than MIPS.

Somewhat surprisingly, the two processors with additional addressing modes (PowerPC and SPARc) execute more memory reference instructions than MIPS, which has one simple addressing mode. If one half of MIPS NOPs are counted as filling load delay slots[1], PowerPC performs 15% more memory operations than MIPS —down from 35% without NOPs. Furthermore, PowerPC executes 7.5× (3.2×) more instructions that load the upper half of a register (MOVEs) than MIPS. The code generator for MIPS seems to be doing an excellent job in generating effective addresses and in keeping data in the global area, from whence it can be referenced via the global pointer `r_gp`.

The compiler seems to fill SPARC's delay slots much more efficiently than it does for MIPS's: the ratio of "exposed NOPs" is 14.7× MIPS/SPARC, or for MIPS 8, 0% of all instructions are NOPs whereas for SPARC these account for 0.44% of all instructions. The numbers for all 27 programs are 16.8× exposed NOPs MIPS/SPARC, 7.2% of all MIPS's instructions ares NOPs, 0.41% of all SPARC's instructions are NOPs. Notice that SPARC ISA only defines branch delay slots whereas MIPS ISA also defines load delay slots.

4.2 Optimization effects

Since the compiler has an important role in performance, we measured the dynamic instructions with the three optimization levels `-O1`, `-O2` e `-O3`. In order to simplify the presentation, we discuss only the results for the encoding and decoding versions of JPEG from CommBench. Table 2 shows, for the two JPEG programs, the ratio of dynamic instructions in the `-O3` version to the other three versions, including the non-optimized `-O0`. An entry *smaller than* 1.00 means the number for `-O3` is *smaller* than that produced with the level indicated for the column. For instance, the MIPS total count for JPEGenc with `-O0` is 1.0/0.37 = 2.7 times larger than the `-O3` version.

Optimized versions with `-O3` display reductions in the number of executed instructions from 2.2× to 2.7× that of the non-optimized versions (`-O0`). The highest gains are in memory references, with elimination of some 3/4 of all references, mainly LOADs. This is relevant because memory references are costly operations and even modest gains might have a large impact on overall performance, not only because of the reduction on instructions executed but also in the better utilization of data that is brought into the cache. The

[1]This is somewhat biased: in JPEGenc there are 2.8× more NOPs in load delay slots than in branch delay slots; for JPEGdec the ratio is 7.5×.

	JPEGenc			JPEGdec		
	`-O0`	`-O1`	`-O2`	`-O0`	`-O1`	`-O2`
	MIPS					
total	0.37	0.95	1.01	0.40	0.96	1.00
ALU	0.75	1.02	1.00	0.92	1.07	1.00
MEM	0.27	1.04	1.01	0.29	1.06	1.00
CTR	0.71	1.00	1.00	0.80	1.19	1.00
FUN	1.00	1.00	1.00	1.00	1.00	1.00
OTH	0.03	0.54	1.01	0.13	0.81	1.01
	PowerPC					
total	0.42	1.04	1.00	0.44	1.02	1.00
ALU	0.56	1.02	1.00	0.70	1.08	1.01
MEM	0.27	1.04	1.01	0.28	1.00	1.00
CTR	0.71	1.00	1.00	0.69	0.91	1.00
FUN	1.00	1.00	1.00	1.00	1.00	1.00
OTH	0.42	2.00	1.00	2.50	2.96	1.00
	SPARC					
total	0.45	1.03	1.00	0.48	1.05	1.00
ALU	0.73	1.06	1.00	0.78	1.07	1.01
MEM	0.24	1.00	1.00	0.29	1.04	1.00
CTR	0.74	1.02	1.00	0.70	1.04	1.00
FUN	1.00	1.00	1.00	0.99	0.99	1.00
OTH	0.03	0.20	1.00	0.00	0.92	1.00

Table 2: Optimization effects compared to -O3, JPEG encode and decode

reduction in the number of NOPs is also worth mentioning: from 7.7× to 33× for MIPS, and from 175× to 1018× for SPARC.

The differences between the optimized versions from `-O1` to `-O3` are on the order of 5%, but some interesting cases are worth mentioning. For MIPS, the reduction in NOPs from `-O2` to `-O3` is between 60 to 70%, and the number of JUMPs in the `-O3` is 7.2× that of the `-O2` version. For SPARC, the number of JUMPs increases in the most optimized versions, by 1.27× and 2.5× for JPEGenc and JPEDdec, respectively. For both processors, the number of JUMPs is under 2% of all instructions executed and therefore these effects are not catastrophic. For PowerPC, in the `-O3` version there in an increase in the number of MOVEs, in the number of logical operations, and also a larger number of load immediates to the most significant half of registers, and these result from more involved computation of addresses and possibly from reducing the strength of some operations.

It must be borne in mind that these figures refer to the *number* of instructions executed and may not be directly related to their 'quality' nor their execution order. By 'quality' we mean the 'latency' or "repetition rate" of multiplies and divides, or references to static data through the global pointer (MIPS). The compiler tries to reorder the code in order to fill delay slots with useful instructions. With superscalar processors, the compiler can group instructions in tuples that have a better chance of parallel execution. These gains can only be assessed with detailed simulation or direct hardware execution.

4.3 Time profile

The time distribution of memory references can help in designing memory hierarchies for embedded systems. The histograms in Figure 2 show the distribution of distances among pairs of LOADs, pairs of STOREs, and pairs of branches. 'Distance' is the number of instructions between the members of the pair, and "zero distance" means two consecutive instructions. All distances longer than 15 instructions are counted on the bar marked '+'. These data were collected for the CommBench programs.

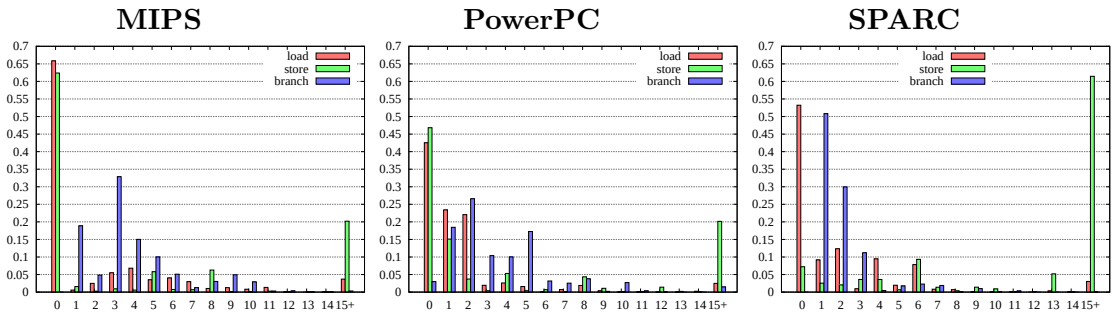

Figure 2: Distance between pairs of loads, stores and branches, all CommBench programs

For MIPS, 80% of all LOADs are separated by less than five instructions, a large majority of 62% of STOREs are adjacent, but 20% are separated by more than 14 instructions. For PowerPC the distribution of LOADs and STOREs is very similar to that of MIPS with smaller peaks. For SPARC, LOADs display roughly the same behavior: 85% have less than five intervening instructions. As for STOREs, less than 30% are clustered, with 62% separated by more than 14 instructions, highlighting the positive effects of the register windows.

These programs display a behavior that can be problematic to a designer of memory interfaces. Since the references are so close together, the CPU-cache and cache-memory interfaces must be designed to deliver near peak bandwidth. With such a high concentration of clustered stores, there must be some form of store queue to accommodate the speed differences between CPU and memory, regardless of how good the cache design may be.

MIPS and PowerPC display a similar distribution of conditional branches, with the majority ($\approx$80%) of the branches separated by up to six instructions. For PowerPC, 2.5% of the branches are adjacent —these may not cause any stalls because the processor is capable of resolving a branch condition in the same cycle it is dispatched for execution. The data for SPARC shows that 90% (ninety) of the branches are separated by up to 3 instructions! The proportion of branches to all instructions is 15.7%. 18.7%, and 20.8% for MIPS, PowerPC e SPARC, respectively.

The distance distributions for JPEG-decode are shown in Figure 3. The three processors show similar behaviors with this program. LOADs are clustered in groups of up to 5 instructions and about half of STOREs are separated by up to 8 instructions —SPARC does not seem to benefit from the register window. The loops that perform most of the work were translated to similar instruction sequences because the distribution of branches are also very similar for the three processors.

JPEG causes relatively high miss rates on the first level of the memory hierarchy [4]. Thus, a cache design for a device in which JPEG is the primary application must be capable of providing high bandwidth between CPU, cache, and memory because of the

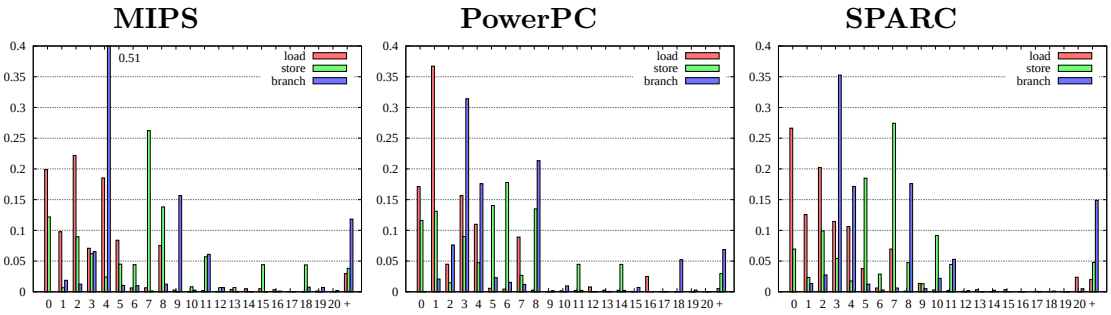

Figure 3: Distance between pairs of loads, stores and branches, JPEGdec

clustering of references and of the high miss rates. As for the control instructions, the distances between branches indicate that a relatively simple branch predictor can provide adequate performance.

5 Conclusion

This paper contains a comparison, at the level of instruction set, of the three 32 bit microprocessors MIPS, PowerPC, and SPARC. The instruction sets were compared by functional simulation of the execution of 16 programs from the MediaBench suite and 11 programs from the CommBench suite. The programs were all compiled with the same version of GCC and were run on the same simulation framework.

The dynamic instruction counts for the 27 programs are 24.9, 27.0, and $26.0 \cdot 10^9$ for MIPS, PowerPC and SPARC, respectively. If memory references are ignored, SPARC has the highest instruction count of the three; PowerPC has the highest memory reference count; MIPS has the better overall performance, contrary to our expectations and in spite of the architectural devices of the two other contenders: register windows (SPARC) and more complex instructions (PowerPC).

We assessed the effects of three levels of optimization (`-O1`, `-O2`, `-O3`) on dynamic instructions and found small differences in instruction counts. We also measured the distribution in time of memory references and conditional branches. While running CommBench programs, about 80% of LOADs and 60% of STOREs are separated by less than five instructions, for the three processors. For MIPS and PowerPC, about 80% of conditional branches are separated by up to six instructions, while for SPARC about 90% of branches are separated by up to three instructions.

References

[1] M C Becker, M S Allen, C R Moore, J S Muhich, and D P Tuttle. The PowerPC 601 microprocessor. *IEEE Computer*, 13(5):54–68, October 1993.

[2] R F Cmelik, D R Ditzel, E J Kelly, and S I Kong. An analysis of MIPS and SPARC instruction set utilization on the SPEC benchmarks. In *4th Intl Conf Arch'l Support for Progr Lang and Oper Sys (ASPLOS'91)*, pages 290–302, Apr 1991.

[3] T Groetker, S Liao, G Martin, and S Swan. *System Design with SystemC*, volume 1. Webman, 2002.

[4] Giancarlo C Heck and Roberto A Hexsel. The performance of Pollution Control Victim Cache for embedded systems. In *SBCCI'07: 20th Conf on Integrated Circuits and Systems Design*, pages 46–51, 2008.

[5] J Hennessy, N Jouppi, S Przybylski, C Rowen, T Gross, F Baskett, and J Gill. MIPS: A microprocessor architecture. *ACM SIGMICRO Newsletter*, 13(4):17–22, December 1982.

[6] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1st edition, 1990. ISBN 1-55860-069-8.

[7] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2006. ISBN 0-12-370490-1.

[8] G Kane and J Heinrich. *MIPS RISC Architecture*. Prentice Hall, 2nd edition, 1991.

[9] C Lee, M Potkonjak, and W H Mangione-Smith. MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *30th Annual IEEE/ACM Int Symp on Microarchitecture (MICRO'97)*, pages 330–335, August 1997.

[10] MIPS. *MIPS32 Architecture For Programmers – The MIPS32 Instruction Set*, volume 2. MIPS Technologies, July 2005.

[11] D A Patterson and C H Sequin. RISC I: A reduced instruction set VLSI computer. In *Proc 8th Intl Symp on Comp Arch (ISCA'81)*, pages 443–457, 1981.

[12] Richard P Paul. *SPARC Architecture, Assembly Language Programming, and C*. Prentice Hall, 2nd edition, 2000.

[13] Power.org. *Power ISA User Instruction Set Architecture*. IBM Corp., October 2007. Book 1 vers 2.05.

[14] G Randin. The 801 minicomputer. In *Proc SIGARCH/SIGPLAN Symp on Architectural Support for Programming Languages and Operating Systems*, pages 39–47, March 1982.

[15] S Rigo, G Araujo, M Bartholomeu, and R Azevedo. ArchC: A SystemC-based architecture description language. In *16th Symp on Comp Arch and High Perf Computing (SBAC'04)*, Outubro 2004.

[16] J Smith and S Weiss. PowerPC 601 and Alpha 21064: A tale of two RISCs. *IEEE Computer*, 27(6):46–58, June 1994.

[17] SPARC International. *The Sparc Architecture Manual – Version 9*. Prentice-Hall, 1994. ISBN 0-13-099227-5.

[18] Tilman Wolf and Mark A Franklin. CommBench – a telecommunications benchmark for network processors. In *Proc IEEE Intl Symp on Performance Analysis of Systems and Software (ISPASS)*, pages 154–162, Apr 2000.

[19] Xilinx. *PowerPC Processor Reference Guide - Embedded Development Kit*. Xilinx Inc., September 2003.