PERL - A Registerless Architecture

P. Suresh Rajat Moona
Indian Institute of Technology Kanpur
Department of Computer Science and Engineering
Kanpur, India
psur@iitk.ernet.in moona@iitk.ernet.in

Abstract

Reducing processor-memory speed gap is one of the major challenges computer architects face today. Efficient use of CPU registers reduces the number of memory accesses. However, registers do incur extra overhead of Load/Store, register allocation and saving of register context across procedure calls. Caches however do not have any such overheads and cache technology has matured to the extent that today the access time of on-chip cache is almost equal to that of registers. This motivates one to explore alternate ways to do away with the overheads of registers.

In this paper, we propose a registerless, memory to memory architecture of a processor. We call this architecture Performance Enhanced Registerless (PERL) processor. All instructions in this processor operate directly on memory operands thus eliminating the Load/Store and other overheads of registers. The performance of this machine is studied by simulations and results are reported in this paper.

1. Introduction

A major challenge for computer architects today is to reduce processor-memory speed gap. One of the time tested mechanism for improving memory system performance is the cache. Current research is also exploring ways such as integration of the memory with logic [12]. This idea is gaining lot of significance since the VLSI technology is projected to put a billion transistors on a single chip.

The two ways to organize the local memory for data are conventional cache, a redundant, dynamically varying subset of memory system – and registers, which are explicitly managed by the program. Benefits from registers are obtained by efficiently allocating them – a process usually done statically by the compiler. Caches are however, independent of architecture and consistently work well taking the dynamic behaviour of the program into account.

The availability of fast high bandwidth on-chip caches and the overheads associated with register allocation, register level context saving, Load/Store associated with registers demand to explore alternate ways to organize local memory. We explore one such method in this paper.

We propose a registerless, memory to memory architecture of a processor and call it Performance Enhanced Registerless(PERL) RISC . It has a simple, small and efficient instruction set and uses pipelined instruction execution to achieve high instruction throughput. All instructions in this processor operate directly on memory operands thus eliminating the Load/Store and other overheads of registers. While a program will execute 30–40% less instructions compared to a normal RISC processor, it is not initialy clear that it will outperform such processors especially the superscalar architectures. We have shown that by using suitable techniques the high bandwidth requirement of such a processor can be met.

The rest of the paper is organized as follows. In section 2, we explain the motivation behind the whole idea and establish the need to investigate. We describe instruction set architecture for the proposed machine in section 3. In section 4, we explain the execution model of the machine and discuss a superscalar model of the machine. A primitive analysis of the machine is done in section 5 comparing it to a typical RISC machine. In section 6, we present the work done so far in this direction. Finally we conclude in section 6, by summarizing the results obtained through simulations and listing various open issues.

2. Motivation

The on-chip and off-chip caches have been the single most technological innovation to reduce the ever growing processor–memory speed gap. The effective access time of on-chip cache with various architectural features reaches close to that of registers. However these two are not equivalent because of the differences in address computation. The registers are small in number and their address is part of

the instruction, whereas the caches form the first level in the memory hierarchy and are not exposed to the programmers. Various tasks associated with cache access, such as TLB access, tag comparison and the actual cache memory access, can however, be pipelined making on-chip cache behave like CPU registers in terms of speed. In fact most of the current day processors are capable of issuing multiple load store instructions in a single clock [5, 13].

Today super-scalar processors make use of multi-ported non-blocking cache to achieve peak performance [5, 14]. Processors currently have large on-chip L2 cache also (secondary cache) to support the misses in on-chip L1 cache (primary cache) [5, 13]. Since on-chip buses can be wider, the bus widths between the processor–L1, L1–L2 and processor–main memory are becoming wider with time. Techniques such as *miss caching*, *victim caching* and *stream buffering* reduce the cache misses [7]. Further, a non-blocking cache allows the service of multiple misses to be overlapped, in a pipelined fashion. Cache bandwidth can be increased by allowing operations such as *load all* which satisfies as many outstanding loads in parallel as possible when data is returned from the cache and *load all wide* which builds on *load all* by widening the single cache port up to the cache block size [16]. Multiple cache ports can be provided either by having multiple copies of cache or by interleaving.

To a certain extent all these techniques to improve cache is to see that the registers get their data fast. We see that in the process the performance of the cache is becoming closer and closer to that of registers. The RISC processors have register to register architecture, which means that all operands for an instruction are in CPU registers except for Load/Store instructions. The program dynamics therefore demands that the operands be loaded into registers and then the computations be performed on them. The underlying memory hierarchy ensures that the operands are also loaded in the cache simultaneously. Temporary use of registers, is however, an exception to this. It is our belief that due to high locality, the operands in registers in most cases will be present in cache also. By removing registers from the hierarchy, especially when on-chip cache are as fast as registers the extra operation of moving operands from cache to registers can be overcome. By not having registers the process of saving register level context is not necessary across procedure calls.

A registerless architecture has other advantages too for a compiler. In conventional processors, arrays, strings and pointers are never allocated to registers. The predominant use of high level languages place the burden of register allocation on compilers making them complex. Registers are not typed, whereas memory operands are typed. Usually the compiler uses extra instructions to type convert data in the register with respect to the data type that it is representing

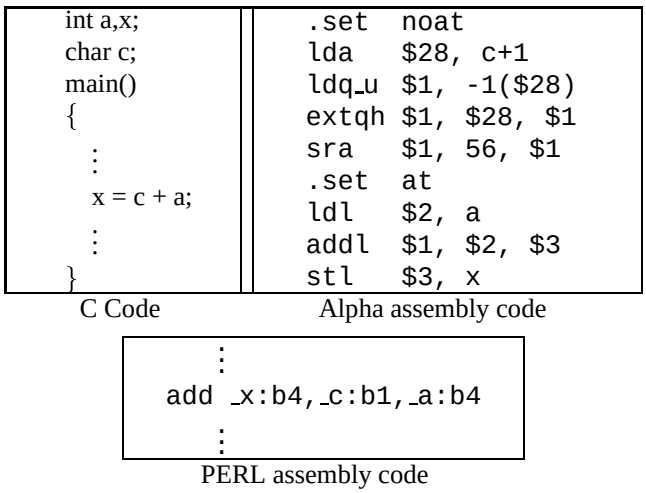

Figure 1. Example

(see Alpha assembly code in figure 1 for example). These instructions include instructions like conversion of a unsigned number to signed number, sign extension etc. In case the operands are taken directly from memory such extra instructions can be eliminated. Further additional shift instructions needed to access unaligned data are avoided when operations are directly performed using memory operands. Any such adjustment can be done on the fly to enable the operation in the execution unit. The example in figure 1 explains this.

In this scenario it is indeed interesting to study a pure memory-to-memory architecture. The instruction length of such a machine will be very long as it has to specify the memory addresses of the operands and destination. At the same time we can expect that programs will execute less number of instructions (about 30–40%) which account for Load/Store, type conversion etc. [6].

We, however cannot get away from registers altogether. The PC will have to remain as a register. Further the stack pointer(SP) and the frame pointer(FP) are used frequently by the programs and access to these have to result always in a hit for good performance. We have found that in a purely memory-to-memory architecture if SP and FP are in memory, they cause 30–40% of the total memory accesses. Essentially SP and FP are used to access the stack variables (the local variables) which are accessed very frequently. In our approach, we have mapped SP and FP onto fixed memory locations which are permanently cached on-chip.

3. Instruction Set Architecture

It is a known fact that the maximum code compaction is obtained in three address format [6]. Further uniform instruction length has its own benefits [6]. These two fac-

tors influenced us to have a three address format. As all operands are in memory, the instruction has to specify three memory addresses as follows.

Operation M1 <:dT>, M2 <:dT>, M3 <:dT>

Here **M1, M2** and **M3** are all in memory. **M1** is the destination of the operation on **M2** and **M3**, whereas **dT** specifies the data type of the corresponding operand.

3.1. Addressing Modes

Addressing modes have the ability to significantly reduce instruction counts. They however, also add to the complexity of building a machine and may increase the average number of clocks per instruction (CPI). There have been lot of studies related to instruction set and its usage [10, 2, 15]. Also there are extensive studies done to compare performance between the instruction set of RISC and CISC [1].

The designers of RISC made extensive study related to the instruction set usage and arrived at the following conclusions [6]. The frequently used addressing modes are *displacement, immediate and register deferred* and these represent 75–99% of the addressing modes used in a program. Further a large percentage of displacement values and immediate values could be represented within 12–16 bits. The memory indirect addressing represents only a small percentage (about 1–16%). Furthermore the PC-relative branch displacement values predominantly could be represented within 8 bits. As a consequence, Loads and Stores in all RISC have the register indirect with immediate index addressing mode **(EA = register + immediate)**, some of them also support register indirect mode **(EA = register)** and register indirect with register index mode **(EA = register + register)** reducing number of instructions per program in certain applications (to the order of 5–6%) [3].

All of the above influenced our design in the following way.

1. All instructions are of the same length.

2. The two operands and destination are each specified by any one of the four addressing methods, namely, *direct*, *indirect*, *displacement* and *immediate*. For displacement addressing the base addresses are certain fixed locations in memory which are permanently cached. Stack pointer (SP) and the Frame pointer (FP) are also part of these fixed locations in memory. The base address is encoded in the instruction using short representations.

3. The simple integer instructions are ADD, SUB, AND, OR, NOT and SHIFT instructions. Integer MULTIPLY and DIVIDE instructions are also provided. These instructions however, have latencies of more than one clock. The same case also holds for the floating point instructions.

4. Jump and conditional branches are supported.

5. The machine supports eight integer data types – byte, half word (16 bit), word (32 bit) and double word (64 bit), each in signed and unsigned flavors. It also supports single and double precision floating point operands.

Since the displacement addressing represents more than one third of the references [6], we decided to provide some fixed locations in the memory to store the base value. These locations are permanently cached assuring 100% hit. One of these locations is used as SP by our compiler. Use of fixed memory locations also saves the instruction space as we shall see. We can extend this idea for temporary variable storage as well.

It can be observed that the instruction length is very long. However, all our instructions will load upto two operands, and operate on them before storing one result. We provide 128 bit wide instructions with space for three memory addresses. If in an instruction both, the operands and the destination, have indirect addressing the processor has to perform as many as 6 memory accesses to execute that instruction. The program locality increases further as there are more memory accesses which otherwise would have been in registers in a conventional processor. The 128 bit instruction space remains under-utilized in case of operations where the number of operands is less than three and also in cases where the displacement and immediate values are small.

The positive aspects of this machine are that it does not have Load/Store instructions. In Reg–Reg machine all operations whose operands are less than the size of the register face additional overheads like sign-extensions, masking etc. Such overheads are not there in PERL processor as the types can be specified in the instruction itself. Further the overhead in context switch is also minimal as the machine state is small (only PC and SP and possibly FP).

4. Execution Model

4.1. Processor Model

The superscalar processor model of PERL processor is represented in figure 2. It consists of an integer and a floating point unit. These operational units are supplied with instruction coming from the instruction queue, where fetched instructions are buffered. Each operational unit contains a set of functional units where instructions are executed, and the results are written to the data cache. In order to support multiple-instruction-issue we have some more elements.

- **multiple out-of-order issue**. The decoder places instructions in program order in the **central instruction window** within the appropriate operational unit. This decouples instruction decoding from instruction execution thereby simplifying dynamic scheduling. The *instruction-issue logic* examines instructions in the window, selects some of them for issue, not necessarily in program order, and dispatches them to their appropriate functional units. There can be any number of instructions active, as long as there are no resource conflicts.

- **multiple out-of-order completion**. Because of the instruction issue policy and various latencies of the functional units, instructions can complete out of program order. Hardware mechanism must ensure that results are written in correct order into memory. Storage conflicts are resolved with operand renaming, using a **reorder buffer** where results of the instructions are placed once computed.

Other mechanism used to accelerate instruction processing and thus enhance the above superscalar features are:

- a **branch target buffer(BTB)** in support of the instruction fetch unit. This enables branch prediction to be performed by the instruction fetch unit. It allows the processor to execute instructions beyond conditional and indirect branches, the reorder buffer is then used to recover from any mis-predicted branch. The instruction fetcher starts fetching instructions from the target address in case of a direct jump.

- a **multi-ported interleaved data cache** is provided to support the multiple instruction fetch and execution. Techniques to improve the cache bandwidth like victim cache, read-all-wide etc. [16], will yield extremely good benefits depending on the address pattern.

4.2. Pipeline Stages

Six overlapping stages or processes make the multiple instruction pipeline, each stage works at its own pace and are explained below.

Fetch Stage: This stage fetches instruction from the cache and places them in an instruction queue. Branches creates two major problems that hinder the fetch mechanism, firstly instruction fetch depends on the outcome of branch execution and secondly the target instruction may mis-align with the cache block, thereby some instructions in the block may not be valid.

The first problem is solved using branch prediction mechanism which uses 2-bit branch history [9]. Indirect jumps are also predicted using the BTB along with a pair of stack similar to the one described in [8]. Direct jumps are handled easily, as their target address can be known at fetch stage itself.

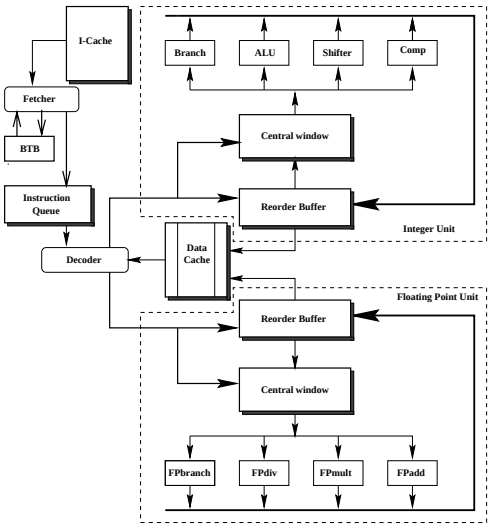

Figure 2. The Processor Model

Decode Stage Here the instructions are taken from the instruction queue, decoded and dispatched to appropriate operational unit. An entry is made in the corresponding reorder buffer for every instruction that is decoded and its destination location identifier is placed in it. The entry is assigned in program order at the tail of the reorder buffer, which is a FIFO queue. To resolve data dependencies, values of the instruction's operands are also placed in the window entry. To do so, the operand address is first searched in the reorder buffer, if it is available and not yet valid (not been computed yet), the corresponding entry is taken in place of the operand value. If the operand value is valid then its value is taken from the reorder buffer. If there are multiple entries of the operand address then the most recent value is taken. If the operands are not found in the reorder buffer then the value is read from the memory.

This stage is implemented as two different stages in the pipeline as the instruction set supports indirect and based addressing modes. For instructions whose operand address mode is indirect or base, the first stage computes the effective address. In both the case there is a memory access to get either the base or the indirect address. There can be upto 3 memory access per instruction. As address forwarding is also done, the reorder buffer is searched in this stage also. The second stage uses the effective address computed in the first stage to read the operands from the memory or takes the reorder buffer tag in case the operand has to be forwarded. There can be upto 2 memory access per instruction.

Execute Stage: The instructions whose operands are available and the required functional unit is available are termed

as *ready* instructions. The issue logic picks up the ready instructions and dispatches to appropriate functional unit. Cases where more than one instruction demand the same functional unit, the oldest of the lot gets the priority. Execute process computes the outcome of the branch instruction and if the prediction is wrong, the instruction following the branch are *flushed* (by marking reorder buffer entries as invalid), and the BTB is updated accordingly. Finally the computed results are written into the reorder buffer.

Write Back Stage: The write back logic finds the completed operations in the reorder buffer and frees the corresponding functional units. The completed results are validated and forwarded to the instructions waiting for them in the instruction windows.

Commit Stage: The validated results are written back to memory during this stage. Writes are processed in order, from the head to the tail of the reorder buffer, until an instruction with an incomplete result is found. The committed instructions are removed from the reorder buffer. Invalidated instructions that follow a mis-predicted branch are simply discarded.

In the worst case the pipeline generates 1 fetch, 5 reads and 1 write access to the cache per instruction. By restricting the base addresses to some fixed location in memory and registering them on the chip will bring down the reads from 5 to 2. Further we can expect the data addresses to have high locality and result in high hit ratio.

5. Analysis

We call the registerless machine as Performance Enhanced Registerless machine **(PERL)**. For our analysis, we use dynamic instruction count from **DLX**, a hypothetical RISC machine [6] for certain reported benchmarks. We make the following reasonable assumptions without giving any undue advantage to PERL.

1. There will be no explicit LOAD/STORE instructions executed in PERL, but will execute the same number of other instructions as in DLX. Further we assume that all ALU instructions in PERL make 6 memory accesses which is the worst case. As opposed to this, DLX is a register to register machine and hence all ALU instructions get executed in a single clock.

2. The cost of execution of branches and the number of dynamically executed branches will be the same in PERL and DLX. Further we assume that cost of branch is one clock, assuming the branch prediction performs with great accuracy. We assume all the branches in PERL are indirect and hence have 2 memory access.

3. The effect of all other instructions are negligible.

From the above assumptions, it is clear that PERL is assumed to have worst case reads. Therefore, the real performance is expected to be better than the analytically arrived one. The actual distribution of dynamically executed instructions for a class of benchmark suite is given in [6] for DLX machine. We can infer from that there are about 25%-45% of Load/Store, about 45%-65% of ALU and about 7%-20% of branches. Now for a program let us assume that on an average there are about 33% of Load/Store, 54% of ALU and 13% of branches.

From the above data we plot the normalized CPI (clocks per instruction) of the two machines for varying hit ratios. The equations 1 and 2 give the CPI of DLX and PERL respectively, normalized with respect to the *instruction count* of DLX. where h is the hit ratio and m_p is the miss penalty in clock cycles. $C_t = 0.13$ if we assume that the branches and other remaining instructions can be executed in one clock cycle each on DLX.

$$\mathrm{CPI_{DLX}} = .54 + .33\,h + .33\,(1-h)\,m_p + C_t \tag{1}$$

$$\mathrm{CPI_{PERL}} = .54 \sum_{r=0}^{6} {}^{6}C_r\, h^r\,(1-h)^{6-r} \cdot (1 + (6-r)\cdot m_p) \\ .13 \sum_{r=0}^{2} {}^{2}C_r\, h^r\,(1-h)^{2-r} \cdot (1 + (2-r)\cdot m_p) \tag{2}$$

Note the missing contribution of load and store instructions in equation 2. Thus equation 2 is the normalized CPI for PERL with respect to the instruction count of DLX . Also note that equation 1 gives CPI for DLX. We can calculate the CPU execution time of the program on both the machines given *N* the total number of dynamically executed instructions in DLX as follows.

$$\text{CPU Execution Time on DLX} = N \cdot \mathrm{CPI_{DLX}}$$
$$\text{CPU Execution Time on PERL} = N \cdot \mathrm{CPI_{DLX}}$$

The miss penalty is same in both cases since both machines are expected to use the same technology for memory. But a well designed second level cache and possibly a third level cache can potentially improve the miss penalty. Further PERL can depend on the compiler to provide effective hints to the cache to improve the hit ratio and avoid unnecessary write backs.

Two performance curves of interest are given in figure 3 and 4. Figure 3 gives the variation of normalized CPI for both DLX and PERL for varying hit ratio. This is significant because it helps to compare the two machines for different hit ratios, as we expect hit ratios will not be the same in the two machines. Figure 4 shows tolerable miss penalty for two cases:

1. When the hit ratios are same for both DLX and PERL.

2. When the hit ratio of DLX is 1.

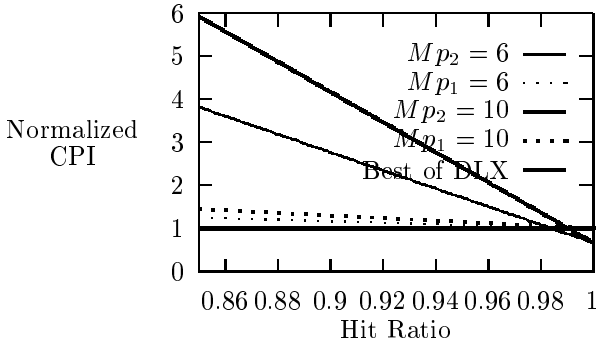

Figure 3. Normalized CPI vs. hit ratio

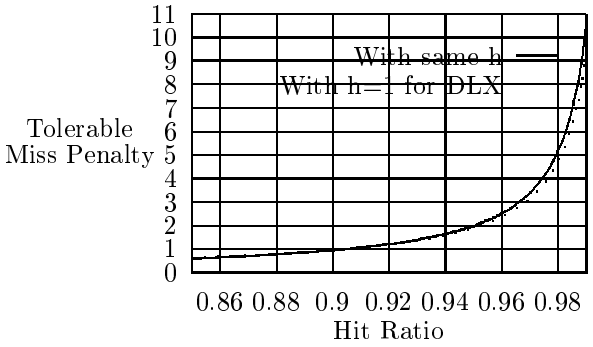

Figure 4. Tolerable miss penalty vs. hit ratio

In figure 4, miss penalties above the curve are the cases where PERL performs better than DLX for the given hit ratio, whereas miss penalties below the curves indicate the cases where DLX performs better.

The curves indicate that as the hit ratio approaches 1 PERL performs better than DLX. Further we assumed the same hit ratio for both PERL and DLX. Intuition points that since the data access pattern will be similar in both DLX and PERL the number of misses will be almost the same in both. So we can hope that PERL will have a higher hit ratio since it generates lot more accesses than DLX.

6. Current Status

There are different ways to evaluate the benefits of design ideas for an architecture, in terms of hardware cost versus performance. We took execution oriented approach and developed a simulator for the PERL processor. This enables us to simulate assembly instructions so that the contents of different hardware elements can be recorded on a cycle basis. Correctness of the program execution can be checked to assess the proper coordination of all different simulated hardware components. The simulator executes assembly programs generated by a cross compiler, so that it can be portable to different machines. The user interface permits one to run the simulator on a cycle basis, and examine the contents of the various hardware elements in a given cycle.

The simulator implements sophisticated superscalar instruction processing policy, multiple out-of-order issue, multiple out-of-order completion etc. Efficient hardware mechanisms like a central window, buffering instruction for issue, and a reorder buffer, operand renaming were selected to achieve the issue policy. Branch prediction to predict both direct and indirect branches is used to keep a steady flow of instructions. Forwarding of data and operand address is also used to further improve the speed up.

In fact the simulator tries to simulate the execution model described in section 4 as closely as possible. It assumes a perfect cache. Various parameters like memory size, number of functional units and their latencies, sizes of the various buffers and the superscalar parameters like number of instruction issue, instruction decode and instruction commit can all be specified in a machine description file.

A C compiler which produces the assembly code is built upon GNU C compiler. GCC gets most of the information about the target machine from a machine description which gives an algebraic formula for each of the machine's instruction. A machine description has two parts: a file of instruction patterns(.md file) and a C header file. Each instruction pattern contains an incomplete RTL expressions, with pieces to be filled in later, operand constraints that restrict how pieces can be filled in, and an output C code to generate the assembler output, all wrapped up in a *define_insn*. We created this file with respect to PERL and used the rest of GCC as it is, to get the compiler for PERL. No machine dependent optimizations are implemented.

A trace driven cache simulator is also built which takes a trace file and a configuration file. The trace file is generated by the processor simulator described above, the traces are in *dinero* format [6]. The configuration contains the description of the cache to be simulated. The cache simulators gives the various performance metrics related to cache.

The complete details of the simulator, compiler and the cache simulator are beyond the scope of this paper.

7. Results

We compare the performance of PERL with that of DLX and DEC Alpha 21064. The compilers for DLX and DEC Alpha are very efficient and perform lot of optimizations, whereas the compiler for PERL does not perform any kind of machine dependent optimizations. The performance figures for DLX were obtained from SuperDLX [11], whereas those for Alpha were obtained using Atom [4]. While simulators for PERL and DLX can be configured to user requirement, Atom captures the performance figures by running on the processor and therefore can not be configured for various architectural variations.

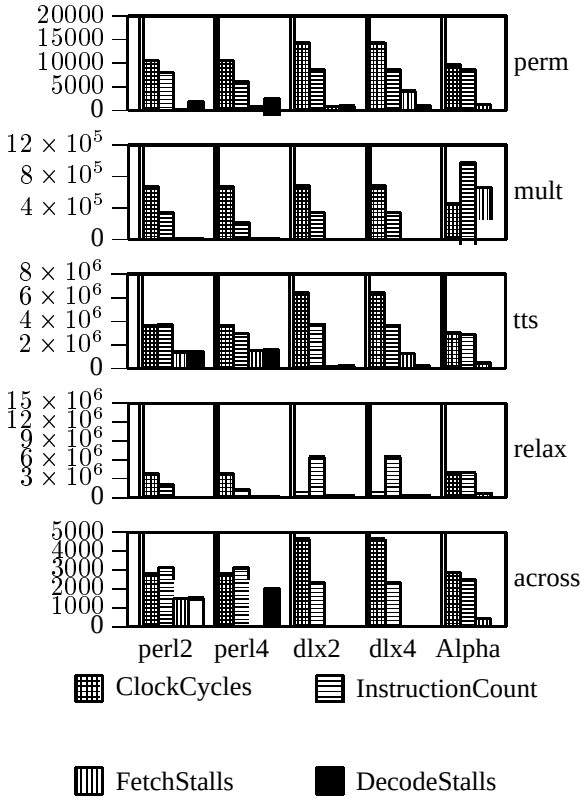

Figure 5. Performance Metrics for PERL, DLX and ALPHA

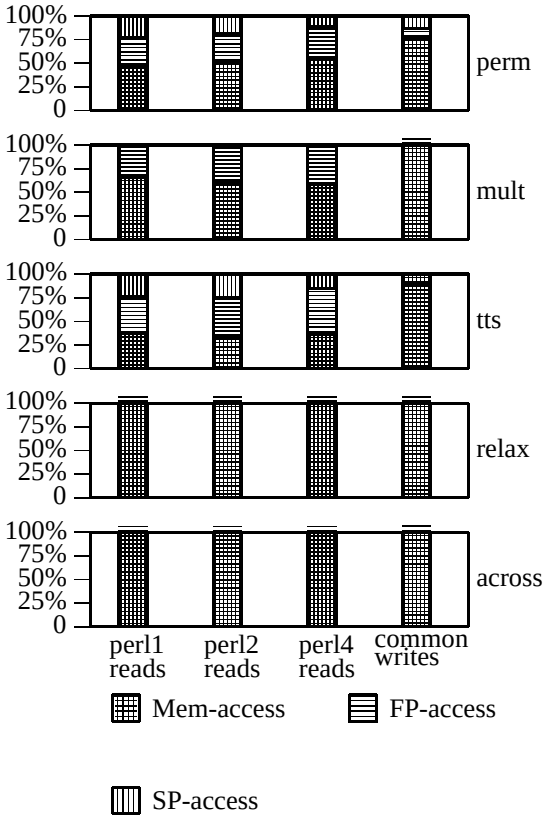

Figure 6. Memory access distribution in PERL

We have taken the following programs from users in our lab as benchmark programs.

perm. This is a heavily recursive program which, given an array of n integers, prints all $n!$ permutations. For simulations we have taken n as 5.

mult. An integer matrix multiplication program. The results are obtained for matrices of size 32x32.

tts. This is a time table scheduler program. Given a list of courses, preferences of timing for allotting slots to the course and a given set of class rooms, the program uses a heuristic driven approach to get the best optimized time table schedule.

relax and across. These two programs are taken from NASA test suite for parallelizing compilers. Both of them contain nested do loops and operate on vectors.

The performance metrics of interest for us are the dynamic instruction count and clock cycles and is plotted in figure 5, for all the programs. The figures are provided for 2-issue and 4-issue PERL and DLX processors and 21064 DEC Alpha processor. As expected PERL executes about 30–40% less instructions, the instruction count in Alpha is smaller than DLX and PERL because it has some special instructions which perform scaled addition, cutting down the instruction required to access array elements.

The total number of clock cycles required to completely execute the program is again better for PERL compared to that for DLX in all cases and Alpha in some cases. The reason why Alpha performs better has to be further investigated.

The Fetch-stalls and Decode-stalls in PERL and DLX are comparable whereas Alpha has lot more stalls. PERL and DLX pipelines have similar designs and various buffers sizes were kept same during simulation, whereas Alpha has fixed buffer sizes.

The access distribution of PERL is interesting because accesses to SP and FP cut down the accesses to cache (see figure 6). It is observed that in case of `perm`, `mult` and `tts` SP and FP accesses contribute 30–60% of the total accesses. Whereas in case of `relax` and `across` benchmarks their contribution is negligible. Both of these benchmarks predominantly use arrays which are allocated on the heap.

We also analyzed the cache performance for PERL and compared it with that of DLX. As there are lot of details associated with the cache design for PERL we cannot present all the results. We have observed the following in case of

PERL.

1. Instruction cache misses for PERL RISC are more compared to DLX. Wider cache blocks however, yield less number of misses in case of PERL.

2. The number of cycles where PERL generates more than 5 memory accesses were very few even for a four issue version.

3. The high data bandwidth requirement of PERL can be satisfied by a dual port interleaved cache. Load all wide technique satisfies about 1–15% of the total accesses.

8. Conclusions

The initial studies indicate that the proposed machine performs better or at least as good as the existing processors. The compiler used is not specifically built for our machine and hence we may be loosing some performance. Some more optimizations that are possible within the scope of the current compiler need to be studied.

The instruction fetch stalls of the proposed machine are considerably high simultaneously the number of cycles in which 2 or 3 instructions are issued is also high when compared with a typical RISC. This may be due to the fact that the compiler is not doing much effort to extract the instruction level parallelism. Improved branch processing by the compiler may also boost the performance. Both these techniques are to be investigated.

The cache performance studies of the proposed machine showed that the absolute number of instruction cache misses are more compared to DLX. But larger block size decreased the number of misses. This is because of the fact the instructions here are long and wider blocks will bring more instructions into the cache and hence reduces compulsory misses. Results also showed that a dual port data cache can satisfy the increased bandwidth requirement of the proposed machine. Further it showed that the data cache performance is comparable or better than the other RISC processors.

This indicates that by having wider block size and wider L1–L2 bus the performance of the proposed machine will be better than that of DLX. This is because the data cache performance is same for both and the proposed machine executes substantially less number of instructions.

Certainly there are some open issues that need to be addressed to substantiate the claim that this is the architecture for the future. The first thing is to estimate the VLSI costs of such a processor. Then the instruction set architecture has to be thoroughly studied to come up with the best set. Compiler optimization techniques specific to this architecture have to be addressed. The proposed machine makes one write in every clock for almost every instruction, the effect of this on the performance of cache consistency protocol has to investigated.

References

[1] D. Bhandarkar and D. W. Clark. Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization. *ACM, Proceedings of the 4th International Conference on ASPLOS*, pages 310–319, 1991.

[2] D. W. Clark and H. M. Levy. Measurement and Analysis of Instruction Set use in the VAX-11/780. *ACM, SIGARCH, Proceedings of the 9th Annual Symposium on Computer Architecture*, pages 9–17, 1982.

[3] R. Cmelik, S. I. Kong, D. R. Ditzel, and E. J. Kelly. An Analysis of MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks. *ACM, Proceedings of the 4th International Conference on ASPLOS*, 1991.

[4] Digital Equipment Corporation. *Atom Reference Manual*.

[5] J. H. Edmondson, P. Rubinfeld, and R. Preston. Superscalar Instruction Execution in the 21164 Alpha Microprocessor. *IEEE Micro*, pages 33–43, Apr 1995.

[6] J. L. Hennessy and D. A.Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, INC, San Mateo, California, 1991.

[7] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the addition of a small Fully-Associative Cache and Prefetch Buffers. *ACM, SIGARCH, Proceedings of The 17th Annual International Symposium on Computer Architecture*, pages 364–373, 1990.

[8] D. R. Kaeli and P. G. Emma. Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns. *ACM, SIGARCH, Proceedings of The 18th Annual International Symposium on Computer Architecture*, pages 34–41, 1991.

[9] J. Lee and A. Smith. Branch Prediction strategies and branch target buffer design. *IEEE Computer*, pages 6–22, Jul 1984.

[10] A. Lunde. Empirical Evaluation of Some Features of Instruction Set Processor Architecture. *Communications of the ACM*, Vol. 20(No. 3):143–153, Mar 1977.

[11] C. Moura. SuperDLX–A Generic Superscalar Simulator. Masters thesis, Advanced compilers, Architecture snd Parallel Sytems Group, McGill University, May 1993.

[12] D. Patterson et al. A Case For INTELLIGENT RAM. *IEEE Micro*, pages 34–44, Mar/Apr 1997.

[13] T. Shanley. *Pentium Pro Processor System Architecture*. Addison Wesley Developers Press, 1997.

[14] M. Tremblay and J. M. O'Connor. UltraSparc I: A Four-Issue Processor Supporting Multimedia. *IEEE Micro*, pages 42–49, Apr 1996.

[15] C. A. Wiecek. A Case study of VAX-11 Instruction Set Usage for Compiler Execution. *ACM, Proceedings of the Symposium on ASPLOS*, pages 177–184, 1982.

[16] K. M. Wilson, K. Olukotun, and M. Rosenblum. Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessor. *ACM, SIGARCH, Proceedings of The 23rd Annual International Symposium on Computer Architecture*, pages 147–157, 1996.