PERL – A Register-Less Processor

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Doctor of Philosophy

by
P. Suresh

to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur
February, 2004

Certificate

Certified that the work contained in the thesis entitled "*PERL – A Register-Less Processor*", by Mr.*P. Suresh*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

(Dr. Rajat Moona)
Professor,
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

February, 2004

Synopsis

Computer architecture designs are influenced historically by three factors: market (users), software and hardware methods, and technology. Advances in fabrication technology are the most dominant factor among them. The performance of a processor is defined by a judicious blend of processor architecture, efficient compiler technology, and effective VLSI implementation. The choices for each of these strongly depend on the technology available for the others. Significant gains in the performance of processors are made due to the ever-improving fabrication technology that made it possible to incorporate architectural novelties such as pipelining, multiple instruction issue, on-chip caches, registers, branch prediction, etc. To supplement these architectural novelties, suitable compiler techniques extract performance by instruction scheduling, code and data placement and other optimizations.

The performance of a computer system is directly related to the time it takes to execute programs, usually known as execution time. The expression for execution time (T), is expressed as a product of the number of instructions executed (N), the average number of machine cycles needed to execute one instruction (Cycles Per Instruction or CPI), and the clock cycle time ( ), as given in equation 1.

$$\frac{\textbf{time}}{\textbf{program}} = \frac{\textbf{instructions}}{\textbf{program}} \cdot \frac{\textbf{cycles}}{\textbf{instructions}} \cdot \frac{\textbf{time}}{\textbf{cycle}} \quad (1)$$

or

$$\textbf{T} = \textbf{N} \cdot \textbf{CPI} \cdot$$

Fundamentally, any effort to improve the performance tries to bring down the execution time. A change in the processor architecture or technology usually affects

one or more terms in equation 1.1. While is the factor of fabrication technology, CPI is dependent upon the architecture and compiler level instruction scheduling.

Traditionally programs are written with a memory model in mind. A pure memory-to-memory instruction set can represent the program without any temporary variables resulting in a maximum code compaction. As instructions can specify operations to be performed directly on memory, compilers can get away with the complex register allocation process. The overhead during procedure call is also minimal. The operands in memory are typed, because that is the way the programs view the operands. When operands are brought into the registers this information is lost as registers have no data type associated with them. To overcome this problem, register-to-register instruction set provide additional instructions like load byte, load word, load signed byte etc. These instructions are not necessary in a memory-to-memory instruction set.

In modern processors, caches provide an access speed close to that of the registers and technology allows to implement caches with multiple ports. Thus, the old notion of registers yielding enormous performance gains may not be valid today. Further, by keeping the instruction set orthogonal with only few and simple addressing techniques, a RISC like memory-to-memory processor can be built. By providing suitable hardware resources such as branch prediction, reorder buffers, multiple functional units, multi-ported caches, etc., a superscalar implementation of the same is possible.

In this thesis, we critically investigate the usage of registers and caches as local memory. We also investigate their organizational benefits. We find that on-chip registers that have a different address space from that of the memory are not the best way to organize local memory. Previous studies have shown diminishing returns with large number of registers. However, technology today enables to have a large number of on-chip registers and use dynamic register renaming to improve performance. We also observe that the performance of on-chip cache scales well with the size. The caches are transparent to the users and exploit program and data locality to bridge the processor-memory speed gap. Further, because of the regularity in their organization, caches yield higher density in implementation than that of the registers

and other logic.

With current technology, it is possible to have high speed, multi-port on-chip caches. With the same technology, it is also possible to have a large number of on-chip registers. We argue that in such a case it is better to map "registers" into memory address space and use them as cache. The programs are written with memory model in mind, and hence a memory-to-memory instruction set is the most natural choice for compilation. We find registers, as used by the processors today, have many disadvantages like load/store overhead, large context, extra instructions to change the type of the memory operands etc. We also note that compilers for ILP (Instruction Level Parallelism) processors while register allocation introduce artificial dependences among instructions that were not there in the original program and hinder multiple instruction issue mechanisms.

The investigations clearly indicate that a better way of utilizing on-chip registers is to use them as the first level in memory hierarchy. We introduce a new technique called on-chip registering of memory locations, where the on-chip register set is mapped on to memory locations. We refer to this as level zero (L0) cache.

To exploit on-chip registering of memory location, we determined that a pure memory-to-memory instruction set is the most suitable one. As there is no need of a separate address space for registers, compilers need not perform register allocation. However, compilers will play a significant role in scheduling instructions for efficient execution on the pipeline. We hope that as there is no register pressure on compilers, they can perform a better job in scheduling.

We name the new memory-to-memory architecture as Performance Enhanced Register–Less architecture (PERL). We also propose a simple, reduced, RISC like instruction set. As the instructions are simple, we show that a superscalar implementation of PERL will be an efficient utilization of the resources.

We first analyze the performance of PERL processor using an analytical model. We take the dynamic instruction count statistics from DLX, a hypothetical generic processor. From these statistics we remove all Load/Store instructions (about 33%) and form the instruction count for the PERL processor. We clearly show in the analysis that at high cache hit ratios (which is also possible due to registers mapped

to the memory address space), the proposed PERL architecture consistently performs better than DLX. The analysis also shows that at higher hit ratios, PERL processor can tolerate high miss penalties.

Simulation studies are carried out to further investigate the performance of the new architecture. A highly configurable instruction set simulator is built with various superscalar features. We have retargeted the GNU C cross compiler for PERL. We have also built a cache simulator capable of simulating several bandwidth improvement techniques. The instruction set simulators for DLX and PERL are used to compare their performance.

We performed simulation over several benchmark programs from NASA test suites as well as from SPEC95. The simulation results clearly show that PERL executes about 30% fewer instructions compared to the DLX for any given C program. In addition, it was seen that PERL requires significantly fewer cycles compared to the DLX to execute a program. The results are also consistent across superscalar processors with degree of 2 and 4, with or without branch prediction. Some programs performed better on DLX than on PERL after machine dependent optimizations are performed on DLX programs. In our compiler, we have not implemented any machine level optimization on PERL. We anticipate that PERL code will perform significantly better when machine specific optimizations are used.

In PERL, a new branch prediction technique using a pair of stacks to predict indirect branches is also used. It shows significant performance improvements in predicting indirect branches, especially for call/return pair of branch instructions.

With our simulations, we noticed that the number of data cache misses for programs in PERL is more than that of DLX programs. But as PERL performs more memory accesses than DLX, the miss ratios are better in PERL. This is in tune with our expectation. However, we feel compiler techniques can further help in bringing down the cache misses.

The thesis concludes that a memory-to-memory architecture is an attractive and viable way to improve the processor performance using the existing processor technologies.

Acknowledgements

It is a great pleasure to thank my thesis supervisor Dr. Rajat Moona for his help, encouragement and support throughout the course of this thesis work. But for his insightful comments and guidance, I would not have finished this work successfully.

I also wish to express my thanks to Dr. Deepak Gupta for his kind advises towards the later part of my work. I also express my sincere gratitude to Prof. Somenath Biswas, Prof. Harish Karnick and Prof. Pankaj Jalote for their encouragement, co-operation and support.

I also thank my teachers Dr. S. K. Agarwal, Dr. Dheeraj Sanghi, Dr. T. V. Prabhakar and Dr. R. Sangal. I thank Dr. S. Saxena, Dr. R. K. Ghosh, Dr. P. Gupta, Dr. R. M. K. Sinha, Dr. A. Jain, Dr. S. Ganguly and other faculty in the department for their co-operation and kindness.

Equally important is the role of my beloved parents who have always stood by me. They never expected anything from me and I feel that there is nothing more one can expect from the aging parents. I also thank my elder brother Nagesh and sister Ambika for taking my responsibilities at home.

I also acknowledge my in-laws, who like my parents did not expect anything from me.

I want to thank my wife Mamatha for her support and for taking the responsibility of home without any complaints, especially during my frequent long absence from home. I also like to thank my two kids Jatin and Hithan, with whom I was not able to spend as much time as they would have wished.

I thank the management of my college P.E.S.C.E., Mandya, for having sponsored me to this program. My special thanks to the Principal Dr. B. Chandrashekar for his kind cooperation, especially after my three years of study leave were over.

My thanks also go to Mr. T. Vasudev, Dr. S. Murli, Dr. K.R. Anand Kumar, for their cooperation and help during my absence there. My special thanks also go to Dr. B.G. Prasad. My sincere gratitude goes to all my colleagues in the department. I express my sincere gratitude to the department staff, for their cooperation. My special thanks also go to Mr. Bore Gowda and Mr. Basavaraj for their cooperation and affection.

My sincere thanks to Dr. M.S. Shivkumar for his guidance and support. I express my gratitude to Dr. B. Ramachandra, for his advises and affection he has shown to me. My thanks also go to Dr. P.S. Puttaswamy and late Dr. C.L. Puttaswamy for their advises and affection.

I also spent some pleasurable time with Prof. Subramanya, Mrs. Prabha Subramanya, Prof. Gangadhariah, Mrs. Kamala Gangadhariah. Prof Sathyamurthy, Mrs. Sathyamurthy, my special thanks go to all of them. I had special relationship with Dr. Raghavendra, Shantakka, Dr. T.K Chandrashekar and Ashaji, I express my sincere gratitude to them.

I also thank Dr. Puttaraju and Mrs. Roopa Puttaraju for their kind hospitality and frequent lunches/dinners at their home.

I will never forget the time I spent with Arvind Galagali. I enjoyed the company of Srikanth P.C (lallu), Praveen, Shyamsundar, Shyama, Sriharsha, Govinda, Sridhar, Fedriks, Subramanya, Rajnish, Sunil, Kamath, Girish, Gopi, Srikar, Prasanna, who all were juniors to me and had sought my advise many times and hope I have not disappointed them, they were all cheerful and at times have helped me also.

I also thank my Friends during my stay in SBRA Dr. Harish, Dr. Vivek Mudgil, Dr. Manjula, S. Mishra, K. S. Grover with whom I and my wife had good time.

I also thank G. V. Ramana Kumar, T. S. Balaji and S. K. Bhatnagar for their contributions in this work.

I enjoyed the company of Dr Veena Bansal, Dr. M. M. Gore, Dr. S.V. Rao, Dr. Sajith, Mr. Kshitiz Krishna and Mr. Rajiva. I shared a special relation with Dr. Atul Kumar, especially after 1997 and I cherish every moment of it. I also thank my new friends in Ph.D room Alpana, Vijay Saradhi and K. V. Arya.

I enjoyed the great time I had with Lamba, Santosh, Narayana and others in

Hall 4. I thank all of them for the nice company that they gave me.

I cannot forget to thank Ashok Kumar Bhatt and Praveen Mahapatra for their kind assistance and hospitality during my stay in Hyderabad.

I also thank Mentor Graphics India, Hyderabad for allowing me free access to office space during my visits to meet Dr.

Dedicated

to

my beloved parents

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Excellent processor performance is the result of a judicious blend of smart processor architecture, efficient compiler technology and effective VLSI implementation. The choices for each of these strongly depend on the technology available for the others. Microprocessors are widely used today as Central Processing Units (CPU) in all types of computers ranging from Personal Computers (PC) to large Shared Memory computers. Significant gains in the performance of processors are due to the ever-improving fabrication technology that made it possible to incorporate architectural novelties such as pipelining, multiple instruction issue, on-chip caches, registers, branch prediction, etc. To supplement these architectural novelties, suitable compiler techniques extract performance by techniques such as instruction scheduling, code and data placement.

1.1 Overview of Microprocessors

Computer architecture designs are influenced historically by three factors: market (users), software and hardware methods, and technology. Advances in fabrication technology is the most dominant factor among them.

Chip fabrication technology has improved almost unimaginably over the past two decades, since much of the early research on RISC (Reduced Instruction Set Computers) [1]. The number of transistors on a CPU chip in 1980 was in tens of

thousands. Hardware designers today are packing about 200 million transistors on a single chip [2, 3, 4, 5].

The performance of a computer system is directly related to the time it takes to execute programs, usually known as execution time. The expression for execution time (T), is expressed as a product of the number of instructions executed (N), the average number of machine cycles needed to execute one instruction (Cycles Per Instruction or CPI), and the clock cycle time ( ), as given in equation 1.1.

$$\frac{\textbf{time}}{\textbf{program}} = \frac{\textbf{instructions}}{\textbf{program}} \cdot \frac{\textbf{cycles}}{\textbf{instructions}} \cdot \frac{\textbf{time}}{\textbf{cycle}} \tag{1.1}$$

or

$$\textbf{T} = \textbf{N} \cdot \textbf{CPI} \cdot$$

Fundamentally all efforts to improve the performance try to bring down the execution time. A change in the processor architecture or technology usually affects one or more terms in equation 1.1. While is the factor of fabrication technology, CPI is dependent upon the architecture and compiler level instruction scheduling. Some important architectural features of processors today are instruction set, pipeline, Instruction Level Parallelism (ILP), etc.

1.1.1 Instruction Set: RISC or CISC - A non-issue

The evolution of instruction set can be classified into 3 phases, the CISC (70's and early 80's), the RISC (late 80's and early 90's), and the POST-RISC (late 90's till date). In the early days of microprocessors the machines were all CISC (the term did not exist then; it was coined to contrast with the RISC in mid-1980's) and instructions took multiple and variable number of cycles to execute. Because of change in technology, many architectural changes were made possible and it resulted in improvement of the speed of individual instruction execution. In time, designers could fit enough transistors on the chip to make the most-used instructions in the instruction set execute in a single machine cycle. RISC machines were first to execute instructions in a single cycle but the CISC machines, mostly from Intel and

Motorola, were there right behind. In the POST RISC era today, the challenge is to exploit the ILP and execute more than one instruction in a single clock.

The design philosophy during the early days of microprocessor (70's and early 80's), i.e., the period during which CISC architecture evolved can be characterized as follows.

- reduce the amount of storage used (memory was small, slow and expensive).
- reduce the number of load and store operations (number of registers was small and hence it was better to access operands directly from memory).
- support compatibility by making sure that new processors could execute the existing code.
- have high level software functionality in the hardware to reduce "semantic gap".
- support instructions that will allow assembly language programmers to be creative in being able to write efficient codes.

The state-of-the-art VLSI technology, storage/memory technology, and compiler technology during 70's and early 80's had significant role for CISC designers to adopt the above philosophy. In order to meet these goals, CISC designs supported complex instructions. These complex instructions had variable length and took multiple and often variable number of clock cycles to execute [6]. Each instruction was designed with the maximum possible functionality in order to minimize the number of instructions that a program needed. Control units were mainly microprogrammed because large and complex instructions made it difficult to implement them in hardware. Variable length instructions were implemented in order to reduce code-size and storage used. Higher functionality was used to reduce the number of storage accesses. The processors supported, memory-to-memory, memory-to-register, register-to-memory and register-to-register operations to provide maximum flexibility to the compiler writers. Memory-to-memory operations were supported to reduce the number of instructions (N in equation 1.1). To reduce the memory traffic and

to improve execution speed, they also supported register-to-register and register-to-memory operations. The processors supported a variety of addressing modes to access operands. Assembly language programming was very common, and the use of high level languages (HLLs) was not as dominating as it is today. Compilers just converted the expressions in HLL into equivalent assembly language code, and if one required an optimized code, writing it directly in the assembly language was the only choice. Hence, the strategy employed was "to move complexity from software to hardware".

CISC design philosophy tried to improve performance by keeping the number of instructions per program (N) in equation 1.1 low. The programmer/compiler was expected to achieve this by optimizing the code using appropriate complex instructions to close the semantic gap. Because of its complexity, instructions took multiple and variable number of cycles to execute resulting in a high CPI and for the same reason the processor also had longer clock time.

At this time, Patterson and others began to question whether implementing all of these complex, elaborate instructions in microcode was really the best use of limited transistors [7]. They argued against continuing CISC philosophy and advocated to move complexity from hardware to software.

The transition from CISC to RISC was a radical change in architecture. Instruction sets were changed, sacrificing binary compatibility for performance [8]. RISC class of processors grew out of research on the usage of various class of instructions by compilers and also their dynamic profile [6, 9, 10]. The problem of semantic clash made most complex instruction sets less usable by the compilers [11]. Semantic clash is due to the problems caused by some HLL statements that seemed to be functionally the same as the machine instructions but were actually different in subtle ways. The concluding evidence was that CISC programs used complex instructions infrequently (about 20%) than simple instructions (about 80%). These findings gave way to reduced instruction set adopting "make the common case fast" principle. RISC processors provide simple register-to-register instructions except for load and store instructions, which move data between memory and registers. A simplified instruction set also led to better utilization of hardware techniques such as pipelining and

hardwired decoding. The silicon space vacated by microprogram gave room for more on-chip registers and caches. Smart compiler techniques were employed to keep the frequently used local variables in registers resulting in reduction in memory traffic and improved execution speed.

RISC design philosophy tried to reduce the time per program (T), by decreasing the CPI. The reduction in CPI was achieved by having simple and reduced instruction set, employing hardwired control and pipelined execution. However, the use of simpler instructions by the compilers took the instruction count (N) a little high. The use of simple instructions, pipeline and fabrication technology all contributed to keep the clock cycle time low. Compilers employed smart techniques to allocate registers and schedule instructions to obtain maximum performance from the pipeline.

Early 90's witnessed a convergence of microprocessor architectures, primarily due to advances made in the VLSI technology. Technology allowed designers of CISCs to adopt some features associated with the RISC [12, 13]. Using suitable decoding techniques current CISC processors are able to break complex instructions into sequence of simple micro operations inside the processor. Most of these micro operations are similar to RISC instructions and are executed using a pipeline. RISC processors in turn have enlarged their instructions by adding floating point and multi–media instructions, apparently to improve performance [14, 15]. Many instructions in the current generation of high performance processors bear no resemblance with original RISC and are called POST-RISC processors [8].

One hallmark of RISC CPUs is that the majority of their instructions are hardwired. This is becoming more and more true of traditional CISC machines as well [16]. The IA-32 processors today, execute the bulk of the instructions in a single cycle [17, 18]. On the other hand, RISC CPUs have floating point and SIMD (Single Instruction Multiple Data) instructions that require multiple cycles. The real challenge today is to make more than one instruction execute in a single cycle. Both RISC and CISC processors today are superscalar processors [2, 3, 4, 5, 18, 19], and they issue multiple instructions in a single clock. The advances in technology

have made it possible for these processors to use complex techniques like register renaming, dynamic scheduling, branch prediction etc., in order to improve the performance. These techniques are highly complex and against the original RISC philosophy of keeping processor implementation simple.

An instruction set can be better than the other in many ways. An orthogonal instruction set, in which all the instructions are generally of same length makes code generation easier for the high-level language compilers. Further, if all instructions are of same length, the process of decoding and early detection of branch and jump instructions is simplified, resulting in improvement in the overall code execution speed. It also makes instruction prefetch simple and efficient as the instructions do not extend beyond cache line boundaries. The issues of number and complexity of individual instructions are (these days at least) close to irrelevant.

1.1.2 Instruction Level Parallelism (ILP)

Instruction level parallelism is exploited by the instruction execution techniques that cause individual machine operations, such as memory loads and stores, integer arithmetic, and floating point operations to execute in parallel [20]. Hardware and software techniques are employed to extract ILP from programs written with sequential execution semantics. The techniques employed to exploit ILP are largely transparent to the users. Superscalar processors employ complex hardware techniques like dynamic scheduling and register renaming to extract ILP. In order to gain maximum benefits out of these hardware techniques, software techniques like software pipelining and trace scheduling are employed to expose more ILP. Very Long Instruction Word (VLIW) processors take advantage of ILP to reduce the number of instructions (N). A single instruction in VLIW processor specifies more than one concurrent operation. The high performance is achieved only when the software is able to pack the collection of concurrent operations into instructions [21].

Pipeline

Pipelining is a technique to exploit ILP and is used to break up the execution of each instruction into several steps, each step performed by a separate circuit called pipeline stage. Pipelining has been used in computers since many years now, starting from the IBM 7030 - Stretch in 1961 [22, 23]. Pipelining is the key implementation technique used in the CPUs today and is applied to instruction execution, memory references and floating point operations.

Performance from pipeline is maximized if the pipeline stages are always kept busy. Several situations during program execution prevent the processor to keep the pipeline busy. The execution path on a branch can be determined only after its evaluation and hence processor has to stall until the outcome of branch execution is known (control hazards). Data dependences between instructions in the program force the pipeline to stall. Long latency instructions like load/store, multiply, divide etc., cause resource conflicts resulting in pipeline stalls (structural hazards). Branch prediction and delayed branches are techniques used to reduce the effect of control stalls. Register renaming and data forwarding are used to lessen the effect of data stalls. Multiple functional units are used to reduce the number of resource conflicts.

The IBM 360/91 [24] computer employed pipelining techniques to a great extent and provided dynamic instruction issuing mechanism, known as *Tomasulo's algorithm* [25] after its inventor. It could sustain only one instruction per cycle and was not superscalar, but the strong influence of Tomasulo's algorithm is evident in many superscalar processors today.

Superscalar Processors

Superscalar processing is the ability of the processor to issue multiple instructions during the same clock cycle. Initially viewed as an extension of RISC technology, today superscalar methods are applied to a spectrum of instruction sets. Prominent examples of superscalar processors are DEC Alpha, ARM, MIPS, PowerPC and Ultra SPARC [26, 4, 27, 15] for RISC and Intel Pentium and Itanium for CISC [17, 18]. Several variations of superscalar processors are possible. A super-pipelined machine issues one instruction per cycle, but the cycle time is set to much less than

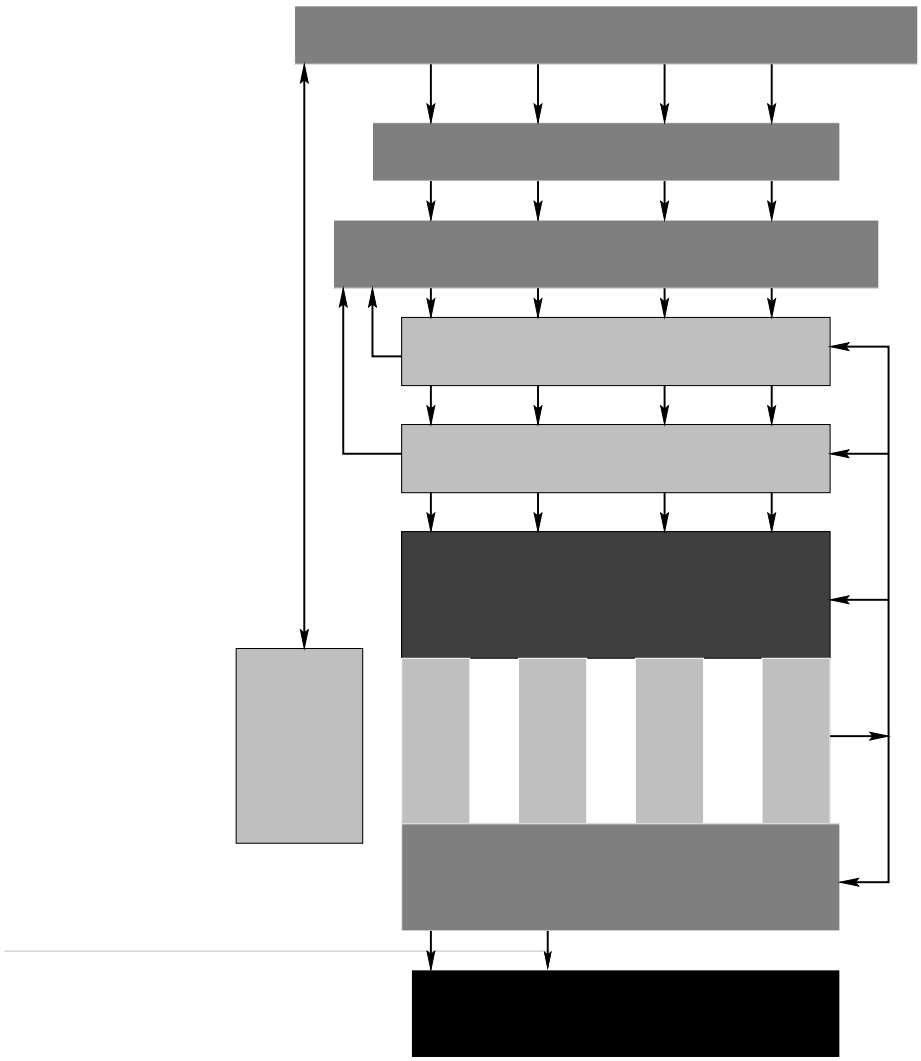
the typical instruction latency [28, 29]. A VLIW machine is similar to a superscalar machine, except that parallel instructions must be explicitly packed by the compiler into very long instruction words [30]. Intel's first 64-bit CPU Itanium's micro-architecture is based on EPIC (Explicitly Parallel Instruction Computing), which is an extension of VLIW technique [31].

Figure 1.1 shows the structure of a generic superscalar processor. A fixed number of instructions are fetched from memory, and augmented with a few characteristic bits in the I-cache. These additional bits identify the type of the instruction such as branch, memory reference etc., and type of execution unit required. The pre-decode technique improves the performance by reducing the complexity of the regular decode stage. The Fetch/Flow unit bring instructions from I-cache using a program counter (PC).

Instructions are decoded and more accurate branch prediction is performed in the decode unit. Most processors use large Branch Target Buffer (BTB), to maintain branch history and to predict the outcome of a branch instruction. If the prediction is later found to be wrong, all results of speculatively executed instructions beyond the branch are discarded. Current processors can simultaneously handle more than one unresolved branches and the corresponding speculative executions.

The superscalar processors place a very high demand on bandwidth of the memory subsystem. With Memory speed not improving at the same rate of processor, superscalar processors have to address the growing processor-memory speed gap [32, 33]. Large multi-ported on-chip caches and large register sets are some of the techniques used to mitigate this problem. Use of on-chip instruction cache, on-chip cache bandwidth and branch prediction has effectively satisfied the instruction bandwidth of ILP processors. However, the same is not true for the data, as the working set of data shows less locality than that of instructions.

There are several studies regarding the available ILP in programs [28, 34, 35, 36, 37]. The limited size of the register set, inherent data dependences that exist in usual program, branches and memory latencies all become a hurdle to maximize the ILP. Studies have indicated that on an average the available ILP in programs is about two [37, 28]. The studies have also indicated that in order to be able to



extract more ILP, compilers require to look beyond basic blocks [34]. The register allocation plays a very important role in such a compiler technology.

1.2 Variable Naming

A program can be viewed as a sequence of instructions that take a set of values as input and produce another set of values as output.

The variables stored in memory are brought into faster caches during program execution. This renaming of memory operands into cache locations is transparent to the programs. The mapping of memory addresses into cache addresses is done by the hardware. Cache just holds small subsets of memory address space.

On the other hand, the variables may be stored in the CPU register. Registers have a separate address space and are addressed using fewer bits compared to the memory addressing. Compilers manage the use of registers. They decide the values that should be kept in the available registers at each point in the program through a process called *register allocation* [38]. There is a natural advantage in keeping the data in the registers as they provide high speed access to the operand values required by the processor. The allocation of registers to program variables is static and done at compile time.

Traditionally the register allocation was done before the instruction scheduling. The processors had no mechanisms to exploit ILP and had very few registers. The performance could be extracted by good register allocation mechanisms and there was very little to be gained by good instruction scheduling [38]. In the ILP processors, the register allocation prior to the instruction scheduling introduces *anti* and *output* dependences that can constrain parallelism. Several new techniques such as modulo variable expansion [39] have been proposed to improve register allocation and obtain good schedule. Register allocation techniques to improve ILP demand large number of registers. Studies have shown that register allocation policies of compilers do not sufficiently exploit the number of registers and reuse registers for data values [22].

Reuse of registers introduces false dependencies between instructions [40]. Superscalar processors handle this problem by performing dynamic register renaming to break the conflict. As there are small number of registers, renaming and resolving data dependencies are fairly simple [22, 41].

1.3 Benefits in Memory to Memory Instruction Set

Programs are written with memory model in mind. A pure memory-to-memory instruction set can represent the program without any temporary variables giving maximum code compaction. As instructions can specify operations to be performed directly on memory, compilers can get away with the complex register allocation process. The overhead during procedure call is also minimal. In register-to-register machines, memory operands which carry type of the operand (int, char etc.) are renamed into registers without any data type information. Extra operations are sometimes necessary to be performed to adjust the data in the register to correspond to its type in memory. These operations could as well be done on the fly while bringing them from memory and before the execution unit uses them. This can be easily done in a memory-to-memory architecture.

As caches today provide an access speed close to that of the registers and technology allows to implement multi-ported caches, a processor with pure memory-to-memory instruction set is surely possible. Further, by keeping the instruction set orthogonal with only few and simple addressing techniques, a RISC like memory-to-memory processor can be built. By providing suitable hardware resources such as branch prediction, reorder buffers, multiple functional units, multi-ported caches, etc., a superscalar implementation of the same is possible.

In this thesis, we investigate the need for a pure memory-to-memory architecture. We also assess its performance by comparing it with a hypothetical RISC processor (DLX [22]).

1.4 Salient Features of the Thesis

In this thesis, we critically investigate the usage of registers and caches as local memory. We also investigate their organizational benefits. We find that on-chip registers, which have a different address space from that of the memory, are not the best way to organize local memory. Previous studies have shown diminishing return with large number of registers [22]. However, the technology today permits to have a large number of on-chip registers and use dynamic register renaming to improve performance. We also observe that the performance of on-chip cache scales well with the size. The caches are transparent to the users and exploit program and data locality to bridge the processor-memory speed gap. Further, because of the regularity in their organization, caches yield higher density in VLSI than that of the registers and other logic.

With current technology, it is possible to have high speed, multi-port on-chip caches. With the same technology, it is also possible to have a large number of on-chip registers. We argue that in such a case it is better to map registers into memory address space. The programs are written with memory model in mind, and hence a memory-to-memory instruction set is the most natural choice for compilation. We find registers, as used by the processors today, have many disadvantages like load/store overhead, large context, extra instructions to change the type of the memory operands etc. We also note that compilers for ILP processors create artificial dependences among instructions that were not there in the original program and hinder multiple instruction issue mechanisms.

The investigations clearly indicate that the better way of utilizing on-chip registers is to use them as the first level in memory hierarchy. We introduce a new technique called on-chip registering of memory locations, where the on-chip registers are mapped on to memory locations. We refer to this as level zero (L0) cache.

To exploit on-chip registering of memory location, we determined that a pure memory-to-memory instruction set is the most suitable one. As there is no need of a separate address space for registers, compilers need not perform register allocation. However, compilers will play a significant role in scheduling instructions for efficient execution on the pipeline. We hope that as there is no register pressure on compilers,

they can perform a better job in scheduling. Compilers can also plug in hints into the instructions to assist in predicting branches and resolving data dependences. They can also intelligently assist in prefetching data and instructions into the cache.

As registers are to be addressed using memory addresses in the proposed architecture, the instructions become very long. We can provide the associated memory bandwidth by having wide buses between instruction cache and decode unit. We show that the instruction bandwidth required for a four way superscalar processor of this kind can be provided using existing technology. Further, we expect program dynamics will be able to capture frequently used operands in the L0 cache.

We also propose a pure memory-to-memory architecture. We name the new architecture as Performance Enhanced Register–Less architecture (PERL). We also propose a simple reduced RISC like instruction set. As the instructions are simple, we show that a superscalar implementation of PERL will be an efficient utilization of the resources.

In this thesis, we also analyze the performance of PERL processor using an analytical model. We take the dynamic instruction count statistics from DLX, a hypothetical generic processor [22]. From these statistics, we remove all Load/Store instructions and form the instruction count for the PERL processor. We clearly show in the analysis that at high cache hit ratios (which is also possible due to registers mapped to the memory address space), the proposed PERL architecture consistently performs better than DLX. The analysis also shows that at higher hit ratios, PERL processor can tolerate high miss penalties.

Simulation studies are carried out to further investigate the performance of the new architecture. A highly configurable instruction set simulator is built with various superscalar features. We have also built a C cross compiler for PERL using *gcc*. We have also built a cache simulator capable of simulating several bandwidth improvement techniques. A superscalar DLX instruction set simulator is used to compare the performance of PERL and DLX.

We performed simulation over a few benchmark programs including the *go* and *compress* programs of SPEC95. The simulation results clearly show that PERL executes fewer instructions (6% to 65%) compared to the DLX for the benchmark

C programs. In addition, it was seen that PERL requires significantly fewer cycles compared to the DLX to execute a program. The results are also consistent across superscalar processors with degree of 2 and 4, with or without branch prediction.

The performance does not vary much even if we provide the same kind of instruction bandwidth to DLX. The Average ILP achieved by DLX across all benchmark is about 2.1. PERL also exhibits an average ILP of about 1.5 (which include loading operands from memory and storing back into memory). As PERL executes fewer instructions this figures indicates that PERL can indeed perform better.

A new branch prediction technique using a pair of stacks to predict indirect branches is used. It shows excellent performance in predicting indirect branches, especially for call/return pair of branch instructions.

The demand on data caches are reduced by operand renaming and forwarding results to waiting instructions. This technique reduced the data cache read traffic (28% to 58%). A small number of registers used as L0 cache where stack pointer and frame pointer are cached along with other program variables. It is shown that this L0 cache caters to more than 20% of total access. The remaining accesses are served efficiently by a large dual-ported cache. A large L2 cache is required to keep the miss penalty down.

The number of data cache misses in PERL and DLX programs were almost same. This is in tune with our expectation. However, we feel compiler techniques can further help in bringing down the cache misses.

1.5 Thesis Organization

The rest of the thesis is organized as follows.

In the next chapter, we present few related works and discuss their innovations and impact on the architectures. Technical details of many contemporary processor technologies are also presented.

In Chapter 3, we discuss the issues involved in the usage of registers such as the weakness of register allocation performed by the compiler. We also look at the cache and various cache performance enhancements used by current day processors.

We also introduce the novel concept of on-chip registering of memory locations and define a pure memory-to-memory architecture (PERL). An analytical performance model of PERL is also presented comparing it with DLX.

In Chapter 4, we present an example design of PERL. We describe a sample instruction set architecture and the pipeline data path to execute the same. We also describe a superscalar implementation of PERL along with techniques such as dynamic scheduling, branch prediction, data forwarding etc.

In Chapter 5, we describe the simulation methodology used to study the performance of PERL. We describe various tools such as *perlcc* (a cross compiler from gcc), **supersim** (a superscalar instruction set simulator) and *cachesim* (a trace driven cache simulator used to study the cache performance of PERL).

The results of the simulations are presented in Chapter 6. The performance of PERL and DLX are compared by executing seven benchmark programs (from SPEC95 and NASA Numerical Aerodynamic Simulation test suites), on the respective simulators. Apart from the execution time, other factors like impact of branch prediction, fetch and decode stalls, and IPC are discussed. In addition, the impact of operand renaming, address/data forwarding and SP/FP accesses are discussed.

We discuss the results of the off-line simulation of memory subsystem in Chapter 7. The impact of L0 cache in PERL is presented and the performance of L1 and L2 caches in DLX and PERL is discussed.

Finally, in Chapter 8, we conclude this thesis. We also present some improvements and identify compiler optimizations specific to PERL. We also present few ideas worth investigating in future.

Chapter 2

Related Work and Contemporary Technologies

In this thesis we used several techniques and results of many research works carried out elsewhere. We present a survey of those research works. We also present technical features of some important contemporary processors.

2.1 Related Work

There are a number of research studies attempting to improve the processor performance by employing intelligent techniques in using local memory (registers and caches). Most of them have addressed the problem within the framework of existing architectures.

The requirement of large register-set by compilers to generate efficient code in superscalar processors have been addressed by many [42, 43, 44]. The approaches involve splitting the register file or providing a cache of the most frequently used registers and having a large backup store for the full set of registers. Postiff et al., proposed a technique [45], where a large logical register file is cached into a small physical register file. This technique then provides a large register set to the compiler. Similarly some researchers have addressed the issue of register port requirement of ILP processors [46, 47]. Register renaming is implemented in several

different ways in commercial processors and a detailed survey of them can be found in the paper by Sima [48].

Improving the memory system is also a subject of major concern and addressed by many research studies. Austin and Sohi [49] showed that the renaming of registers and memory can open up ways to increase ILP several fold. McNiven and Davidson [50] found that values with long live ranges also have long reuse distances. By using compiler directives to keep these and other dead values out of cache, they were able to use the cache better and achieve 25%–30% reduction in memory traffic. Huang and Shen [51] have analyzed the intrinsic memory bandwidth requirements of ordinary programs. They report that with the microprocessors issuing between one and four instructions, the bandwidth requirement scales linearly with the issue rate. They also report that as issue rate increases beyond eight instructions per cycle, the growth in bandwidth requirement increase much more rapidly.

The memory system design considerations for dynamically scheduled processors are well addressed by Farkas et al. [52]. Sites in his paper [53], discusses a spectrum of ways to exploit large number of registers in an architecture – ranging from programmer-managed cache (large number of explicitly addressed registers, as in CRAY-I) to better schemes for automatically managed cache. A combination of compiler and hardware techniques will be needed to maximize effective register use while minimizing transmission bandwidth between various memories. Discussions include merging activation records at compile time, predictive cache loading, and dribble-back cache unloading. In this paper, he also notes that if caches are as fast as registers then we do not need registers at all, as there is no need of any temporary storage. Davidson and Vaughan in their paper [54], report the results of a set of experiments to isolate and determine the effect of instruction set complexity on cache memory performance and bus traffic. They report that the miss ratio is affected by the object program size and hope that it can be corrected by just increasing the cache size. They conclude that RISC machines require about 4 times larger instruction cache than that in CISC to achieve the same hit ratios as object codes for RISC machines are twice as big as their CISC counterparts. Increasing cache size above 64KB does not provide any additional improvement in performance.

Jouppi [55] proposes three new techniques – miss caching, victim caching and stream buffers, to improve the performance of direct-mapped cache. Miss caching and victim caching techniques effectively reduce the conflict misses while stream buffers reduce the capacity and compulsory misses.

Sohi and Franklin introduce multi-port non-blocking L1 cache to improve the data bandwidth to greater than 1 request per cycle [56]. The multi-port cache is built using duplicate cache banks and interleaved banks in their paper.

Several other researchers [21, 57] have looked at the efficiency of multiple ports in the cache.

Wilson et al., propose three techniques to increase the cache port efficiency for dynamic superscalar processor [58]. The three techniques are load-all, load-all-wide, keep tags and line buffers. Wilson and Olukotun [59], report that an increase in cache port from 1 to 2 increases the processor performance by 25%, from 2 to 3 ports gives 4-5% increase and from 3 to 4 ports gives 1% increase.

Various researchers have worked on alternative ways of organizing and using on chip local memory [60, 61, 62, 63].

Wall in his paper [35] finds that even with impossibly good techniques, average ILP rarely exceeds 7, with 5 more common. Butler et al. [34], exhibits that when all constraints are removed except those required by the program, the degree of parallelism found can be in excess of 17 instructions per cycle.

The case for processor/memory integration (IRAM) in view of growing chip densities and slow DRAMs has attracted lot of interest [64, 65]. The system level implications of processor-memory integration are studied in a paper by Burger, Goodman and Kagi [66].

The only work that has a strong resemblance to our work is the F-CPU project (F for freedom). This is the only work where a memory-to-memory architecture with some twists has been chosen. The latest document [67] from them says that they have abandoned memory-to-memory architecture. They now use Transport Triggered Architecture (TTA). No reason has been given for their decision to abandon memory-to-memory architecture.

2.2 Review of Some Contemporary Technologies

It is important to look into the technical features of the state-of-art processors to know what they offer in order to improve performance. We also look into new innovative ways of processor implementation technique in the form of Crusoe^{TM} processor from Transmeta.

2.2.1 UltraSPARC III and IV

The UltraSPARC-III is the third generation of Sun Microsystem's most powerful microprocessors. The UltraSPARC-III design extends Sun's SPARC Version 9 architecture a 64-bit extension to the original 32-bit SPARC architecture that traces its roots to the Berkeley RISC-I processor. It can sustain the execution of up to four instructions per cycle, even in the presence of conditional branches and cache misses, mainly because the units asynchronously feed instructions and data to the rest of the pipeline. Instructions that are predicted for execution are issued in program order to multiple functional units, executed in parallel, and for added parallelism can be completed out-of-order. To further increase the number of instructions executed per cycle, instructions from two basic blocks can be issued in the same group [15, 68].

The core instruction set has been extended to include graphics instructions that provide the most common operations related to two-dimensional image processing, two and three dimensional graphics and image compression algorithms, and parallel operations on pixel data with 8 and 16-bit components.

Some of the UltraSPARC III processor features are shown in figure 2.1 and listed below.

- 4-way superscalar processor with nine execution units and six execution pipes (2 integer, 2 floating-point, 1 load/store and 1 branch).
- 14 stage, non-stalling pipeline.
- 64-bit data paths including 64-bit ALUs and 64-bit address arithmetic.
- 64-bit virtual address and 43-bit physical address space.

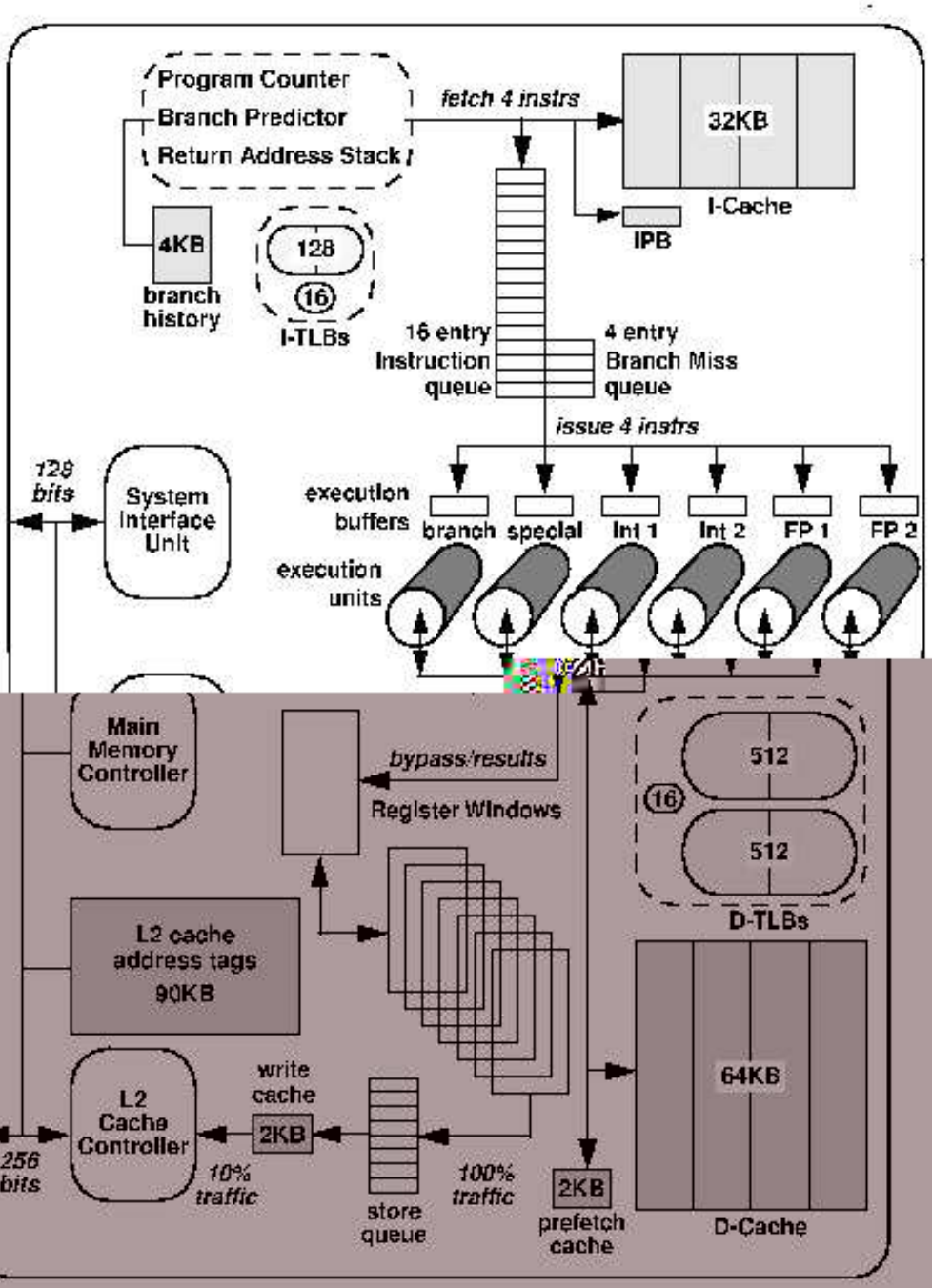


Figure 2.1: UltraSPARC III[1]

[1] < Source: "An Overview of UltraSPARC III Cu Processor", Sun Microsystems, A white paper, Jun 2002 [68] >

- Data and instruction prefetching mechanism.
- Data Memory Management Unit with 1040 Translation Lookaside Buffer (TLB) entries that can support up to 4 MB pages.
- 900 MHz or higher frequency (Currently shipping 1.2 GHz processor).
- Primary instruction cache (e.g., 32KB, 4 way set associative in UltraSPARC III).
- Primary data cache (e.g, 64 KB, 4-way set associative supporting one load or store instruction per cycle in UltraSPARC III).
- Prefetch cache for software prefetch (e.g., 2 KB, 4-way set associative in UltraSPARC III).
- Write cache that reduces store bandwidth to Level 2 cache (e.g., 2 KB, 4-way set associative in UltraSPARC III).
- Support for L2 cache (e.g., 8MB, 2-way set associative external L2 cache in UltraSPARC III).

The data cache is virtually indexed, physically tagged cache (VIPT) with a two cycle latency and one-cycle throughput. It employs write-through, no write-allocate policies. The L1 cache miss penalty is 10 cycles and L2 miss penalty is about 100 cycles.

The UltraSPARC III Cu processor contains 160 general purpose registers. They are windowed into 32 registers addressable by Integer Unit Instructions. The Floating-point register file contains sixty four 32-bit registers and can be addressed as thirty two 32-bit FP registers, thirty two 64-bit FP registers and sixteen 128-bit registers. The details of the register addresses and allocation can be found in [3].

The SUN's fourth-generation Ultra SPARC IV is made up of two UltraSPARC-III cores (code-named Cheetah) and includes on-chip tags for 8 MB of off-chip 2-way set-associative level 2 cache per core. Other shared interfaces on the UltraSPARC IV processor include an on-chip memory controller supporting up to 16 GB of DRAM.

The new chips are part of Sun's "Throughput Computing" strategy, which the company acquired from Afara Websystems in 2001. At the heart of this new strategy is Chip Multi-Threading (CMT), a design concept that allows the processor to execute tens of threads simultaneously. The first generation of CMT, such as the UltraSPARC IV family of processors, will enhance current UltraSPARC III system throughput, initially by up to 2 times, and later by up to 3 to 4 times the current levels. In future, Sun says it will be rolling out a more radical CMT design, which will first appear in Sun's Blade platform in 2006, which will increase the throughput of today's UltraSPARC IIIi systems by up to 15 times.

2.2.2 MIPS R18000

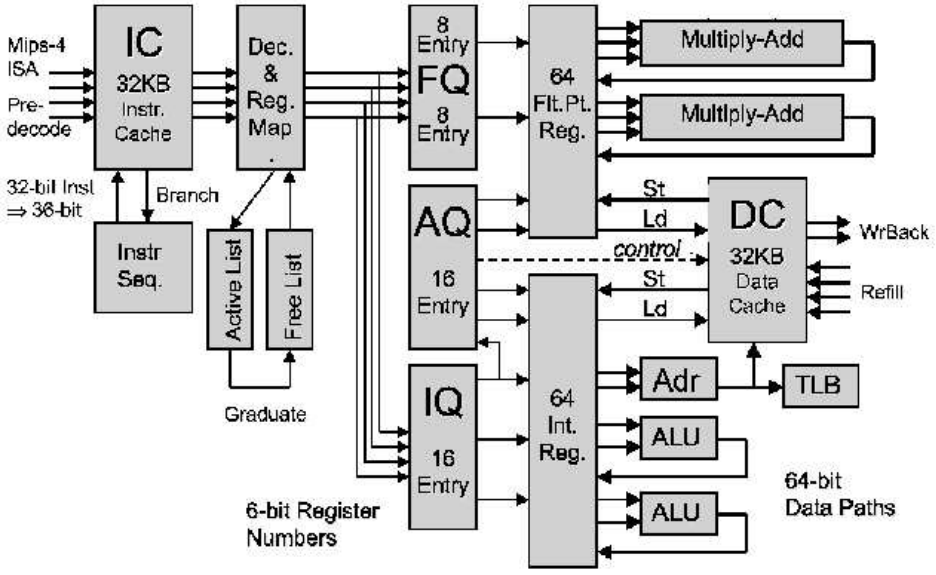


Figure 2.2: The micro-architecture of MIPS R18000[2]

[2]< Source: Fu et al., "R18000^{TM} The Latest SGI^{TM} Superscalar Microprocessor" - Hot Chips 13, Aug 2001 [4] >

The MIPS R18000 is a four-way superscalar processor. It fetches and decodes four instructions per cycle and speculatively executes beyond branches with a four-entry branch stack. It uses dynamic out-of order execution, implements register renaming logic using map tables and achieve in-order graduation for precise exception handling.

It employs both branch prediction and speculative execution. The instructions are executed out-of-order and committed in-order. The processor uses register renaming to reduce the latencies due to false data dependencies. To sustain the four-way issue, there are five fully pipelined, low latency execution units. More details of the performance aspects of the processor can be found in a paper by Fu et al. [4].

The MIPS R18000 can hold up to 48 decoded instruction (active list). This helps in issuing a large number of speculative instructions. Split transfer from system address bus to secondary cache shortens the busy periods and hence provides better utilization of system bus.

The MIPS R18000 has a 2-way set-associative 32KB on-chip L1 data cache and 32KB instruction cache. It has 1MB 4-way set associative on-chip L2 cache. Both L1 and L2 are non-blocking, write back caches and employ LRU replacement policy.

2.2.3 Alpha Architecture

Even though Alpha architecture has not been a very popular architecture, it deployed many novel schemes. The Alpha 21364 is the latest of alpha processor from the Compaq computer corporation. The 21364 use the same core as that of 21264 with some minor enhancements. Compaq had also released the design of 21464 with a completely new micro-architecture [19]. Originally scheduled to be ready by 2002, the chip is yet to be launched.

Some of the features of alpha 21364 (figure 2.3) are as follows.

- 0.18 **m** CMOS, 1250 MHz.
- 152 million transistors (15 million logic, 137 million SRAM).
- Four-issue superscalar processor.

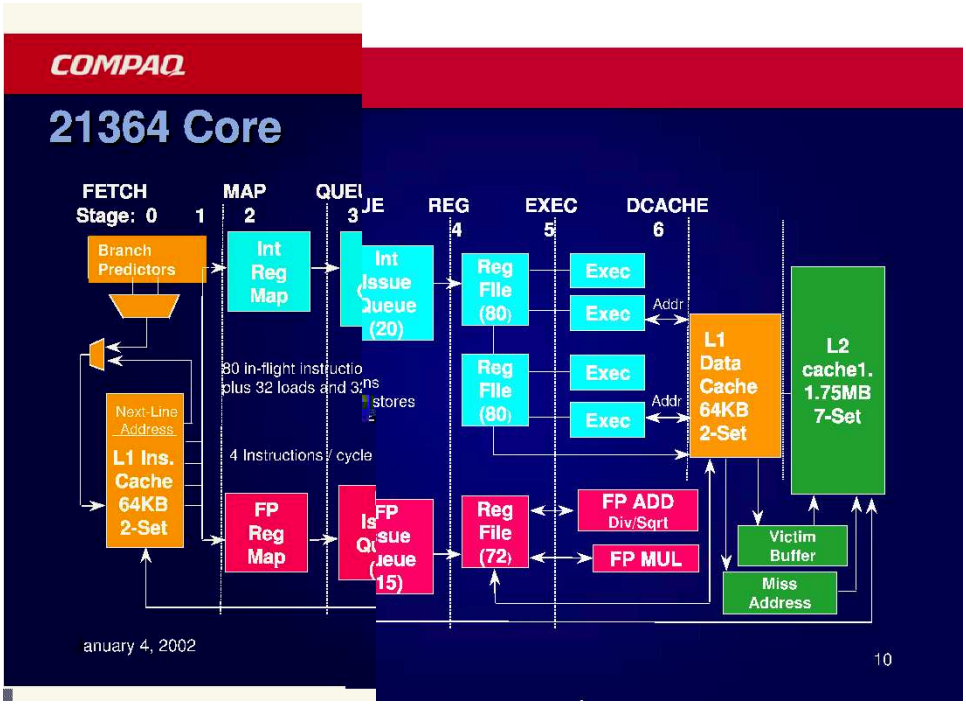


Figure 2.3: The Alpha 21364 core[3]

- 64KB I-cache, 64KB D-cache.
- Integrated L2 cache (1.75 MB).
- Integrated system/multiprocessor interface.
- Integrated memory controller (RAMBUS interface), high data capacity per processor, 800 MHz with 30ns CAS latency pin to pin.
- Integrated network interface (3GB/s I/O interface per processor).
- Supports lock-step operation to enable high availability.
- System-on-a-chip concept.

Alpha 21364 has the alpha 21264 core with 64 KB of instruction cache and data cache. Both of them are 2-way set associative and have a latency of 2 cycles.

[3]< Source: P. Bannon, "Alpha 21364 – A Scalable Single–Chip SMP", Microprocessor Forum, Oct 1998 [69] >

However, they are pipeline and hence the execution engine can issue two loads or one store simultaneously. To support the dual port, the entire data cache is duplicated unlike other processors that have multiple banks. It has an on-chip L2 cache of 1.75MB (7-way set associative), with a huge 12 cycle latency (due to the reuse of 21264 core). The L2 cache offers a total read/write bandwidth of 20GB/s. Data can be accessed in 4 cycle blocks. The 21364 can buffer up to 16 L1 cache miss requests at a time. Similarly, it has 32 victim buffers, which hold dirty cache lines on their way back to memory (16 each for L1 and L2).

The core of the 21364 is an out-of-order processor with a peak execution rate of size seven instructions per cycle and a sustainable rate of four per cycle. The processor can keep up this pace on either integer or floating-point code. Up to 80 instructions can be in processed at a time. Registers are renamed on the fly, with 80 integer registers and 72 floating-point registers. The 21364 duplicates the integer register file, with each copy having four read and six write ports.

The alpha 21464 has a totally new micro-architecture and is designed to have the following features.

- An aggressive instruction fetch unit.
- 8 wide superscalar execution unit.
- 4-way Simultaneous Multi-Threading (SMT).
- Large on-chip L2.
- Direct RAMBUS interface.
- On-chip router for system interconnect.
- Up to 512-way multiprocessing.
- Directory-based Non Uniform Memory Architecture (CCNUMA).

The 21464 has a single-issue queue with 112+ entries and is 8-instruction wide superscalar issue capability. To support this it has features like next address generation, line predictor, branch predictor, jump target prediction and return address stack.

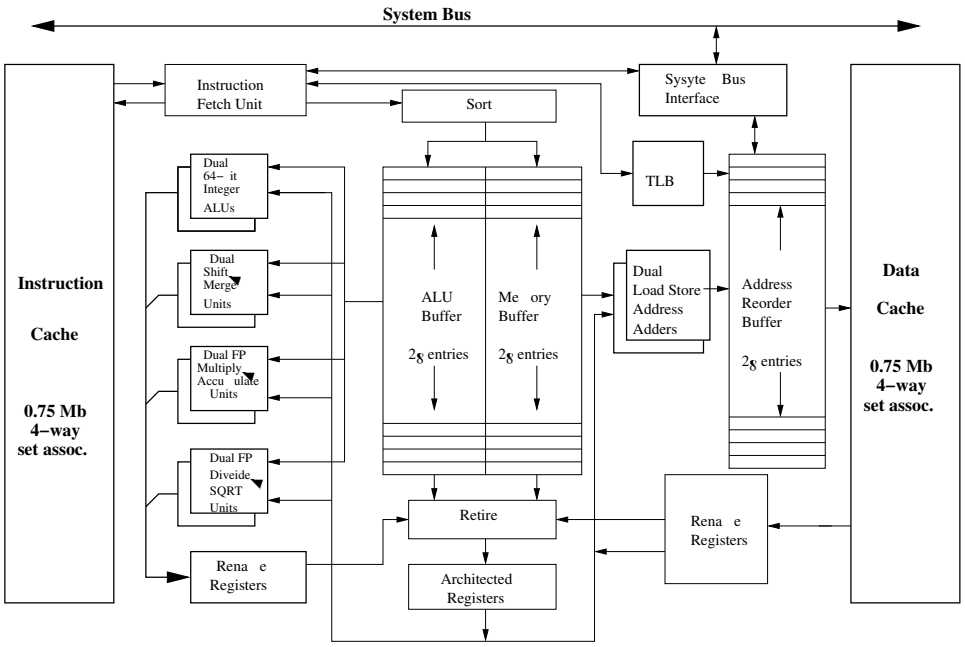


Figure 2.4: Block diagram of HP PA-8700 processor[4]

It has 8 integer and 4 floating point ALUs and 4 memory (2 read and 2 write units). All these are pipelined. To support this wide issue execution engine it has 512 registers with 16 read ports and 8 write ports.

The L1 instruction cache is dual ported and is capable of supplying two eight-instruction bundles per cycle.

2.2.4 HP-Precision Architecture

Hewlett-Packard (HP) over the years has developed a highly successful RISC processor line – the PA-RISC (Precision Architecture RISC) family. The PA-8700 (figure 2.4) is the latest among them with the following features [5].

- 64-bit extensions for both data and addresses.
- Multi-media extensions.
- branch prediction with "hints".

[4]< Source: "PA-RISC 8X00 Family of Microprocessors with Focus on PA-8700", Technical white paper, Hewlett Packard, April 2000 [5] >

- Weak memory ordering.

Some notable features of PA-8700 are the following.

- Clock frequency greater than 800 MHz.
- 1.5 MB of L1 data cache and 0.75 MB of L1 instruction cache (4-way set associative).
- Data prefetching capability.

The advanced micro-architecture of the PA-8700 aggressively executes as many instructions as possible in each cycle to maximize performance. The processor exploits techniques such as out-of-order execution, speculative execution, and non-blocking caches. The Instruction Fetch Unit can fetch 4 instructions each cycle from the instruction cache. From there, the instructions are forwarded to the Sort unit, which places instructions into either the 28 entry ALU Reorder Buffer or the 28 entry Memory Reorder Buffer depending on the instruction type. As soon as an instruction is identified as ready to be executed, and an appropriate functional unit is available, the instruction is dispatched to the functional unit to begin execution. Each cycle, up to four out-of-order instructions can be dispatched. The PA-8700 includes 10 functional units to maximize the number of instructions that can be executed in each cycle. These functional units consist of 4 integer units (2 Arithmetic/Logic Units and 2 Shift Merge Units), 4 floating point units (2 Multiply and Accumulate Units and 2 Divide/Square Root Units), and 2 Load/Store Units (one for even double-word addresses and one for odd double-word addresses). After instructions have completed execution in the various functional units, they return to the Retire Unit to update the architected state of the processor and thus complete execution.

2.2.5 Pentium-4 Processor

The Pentium-4 (P4) is the Intel's state-of-the-art processor based on IA-32 architecture [17]. It uses the NetBurst micro-architecture [70]. Its deeply pipelined design delivers high frequencies and performance. It uses many novel micro-architectural

ideas including a trace cache, double-clocked ALU, new low latency L1 data cache algorithms, and a new high bandwidth system bus.

NetBurst^{TM} Micro-architecture

Intel's NetBurst^{TM} micro-architecture consists of four main sections as shown in figure 2.5

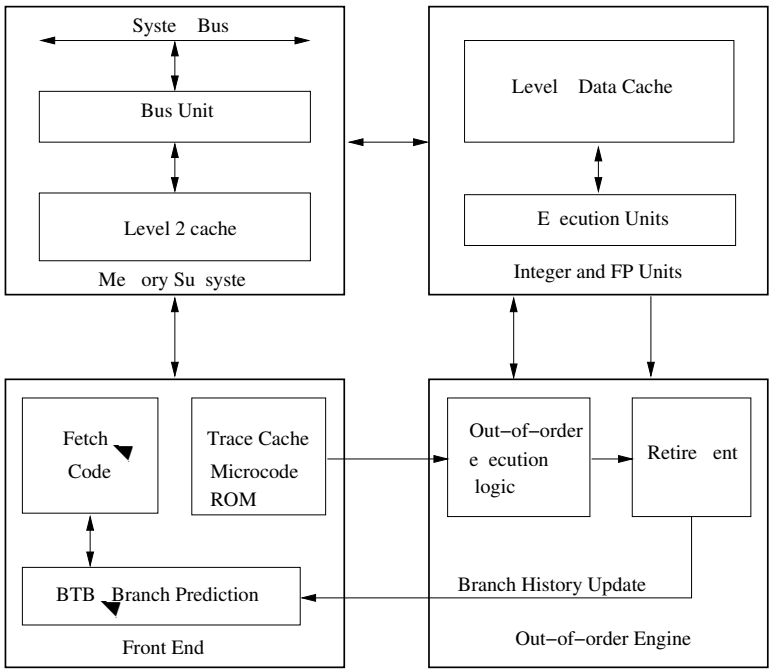


Figure 2.5: Basic block diagram of NetBurst^{TM} micro-architecture[5]

The in-order front end uses very aggressive instruction fetch techniques supported by highly accurate branch prediction logic. These IA-32 instruction bytes are then decoded into basic operations called ops (micro-operations) that the execution core will execute. The NetBurst^{TM} micro-architecture has an advanced form of level 1 (L1) instruction cache called the execution trace cache. The trace cache stores the already decoded IA-32 instructions or ops. The IA-32 instruction decoder is only used when the machine misses the trace cache and needs to go to L2 cache to get

[5]< Source: Hinton et al., "The Micro-architecture of Pentium 4 Processor" - Intel Technology Journal Q1, 2001 [70] >

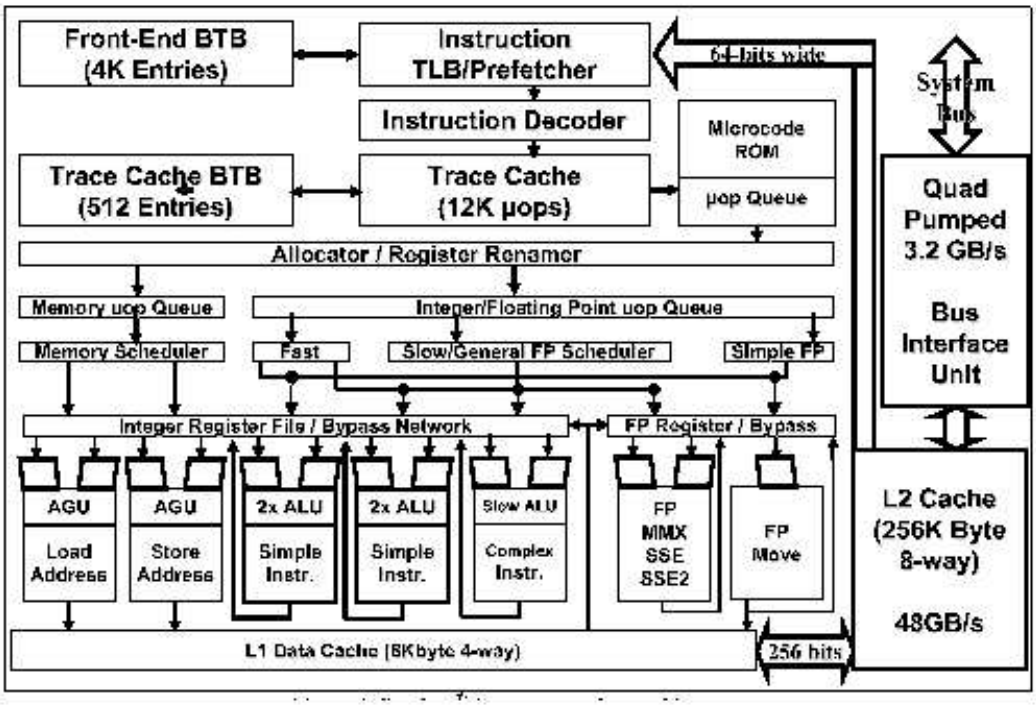


Figure 2.6: Block diagram of Pentium–4 processor[6]

and decode new IA-32 instruction bytes.

In the out-of-order execution engine, instructions are aggressively reordered to allow them to execute as quickly as their input operands are ready. It even executes the instructions in the program that follow a delayed instruction as long as they are independent from the delayed instruction. Register renaming logic renames the logical IA-32 registers such as EAX onto the processors 128-entry physical register file. This allows the small, 8-entry, architecturally defined IA-32 register file to be dynamically expanded to use the 128 physical registers. The retirement logic reorders the instructions executed out-of-order. Pentium 4 can retire up to three ops per clock cycle. The logic also reports branch history information to the branch

[6]< Source: Hinton et al., "The Micro-architecture of Pentium 4 Processor" - Intel Technology Journal Q1, 2001 [70] >

predictors at the front end of the machine.

The L1 data cache in the processor is an 8KB-byte, 4-way set-associative cache with block size of 64 bytes. It uses write-through policy and can sustain one load and one store per clock cycle. For cache misses the data obtained is treated as speculated data and the instructions go ahead with incorrect data. A mechanism called replay is used to re-execute instructions that use incorrect data.

Pentium 4 uses deep pipelining to obtain high frequency. Different parts of Pentium 4 processor run at different clock frequencies. The frequency of each section of logic is set to be appropriate for the performance it needs to achieve.

2.2.6 IA-64 Architecture

Intel and Hewlett-Packard(HP) jointly developed the instruction set architecture for IA-64, with a goal of maintaining backward compatibility with existing IA-32 and HP's Precision Architecture (PA)-RISC software. IA-64 maintains full binary compatibility with IA-32 instructions in hardware and full binary compatibility with PA-RISC instructions through software translation [2, 71].

IA-64 includes features that overcome many of the limitations of IA-32 architecture. In addition to performance improvements, IA-64 has other enhancements.

- **Larger memory addresses:** 64-bit address space can accommodate nearly 18TB of physical memory.

- **Explicit Parallelism:** With the Explicitly Parallel Instruction Computer (EPIC) instruction set developed for IA-64, the compiler plays a much larger role in ensuring parallel execution of instructions. The EPIC compiler analyzes the source code to determine which operations can be executed in parallel, enhances the code to make these operations parallel, and then generates the appropriate machine code.

- **Predication:** The IA-64 uses a technique called predication to handle conditional branches in an efficient manner. The predicates are used by the EPIC compiler. In a simple "`if, then, else`" statement, the compiler generates

straight-line predicated code(no conditional branch) for both true and false paths. At run time, a compare statement stores either a one (true) or zero (false) value in a special predicate register for each branch. Then, the execution unit executes both paths, but only uses the results from the path with the true predicate register. The results of the path with the false predicate register are ignored. The IA-64 architecture includes 64 of these special 1-bit predicate registers.

- **Speculation:** The IA-64 architecture includes mechanisms that permit the compiler to direct control (instruction) and data speculation to reduce or eliminate latency in certain types of operations.

- **Bundled Instructions:** In the IA-64 architecture, instructions are packed in 128-bit bundles to facilitate parallel execution.

- **Byte Ordering:** IA-64 architecture supports both little-endian (low-order bytes first) and big-endian (high-order bytes first) byte ordering for loads and stores longer than 1 byte. A bit in the user mask register controls the processor endianity.

Itanium Processor

Itanium is the Intel's first 64-bit microprocessor (figure 2.7) based on IA-64 architecture. Itanium features 14 port 128 integer registers, 128 floating-point registers, 64 predicate registers, and a large number of special-purpose registers. The CPU core is made up of 25.4 million transistors whereas the L3 cache consists of 295 million transistors.

Itanium features a three-level cache hierarchy with L1 and L2 cache on the microprocessor die and a large 4-MB L3 cache connected to the microprocessor through a dedicated 128-bit bus running at full processor clock speed. The L1 cache includes a separate dual ported 16KB data cache and 16KB instruction cache.

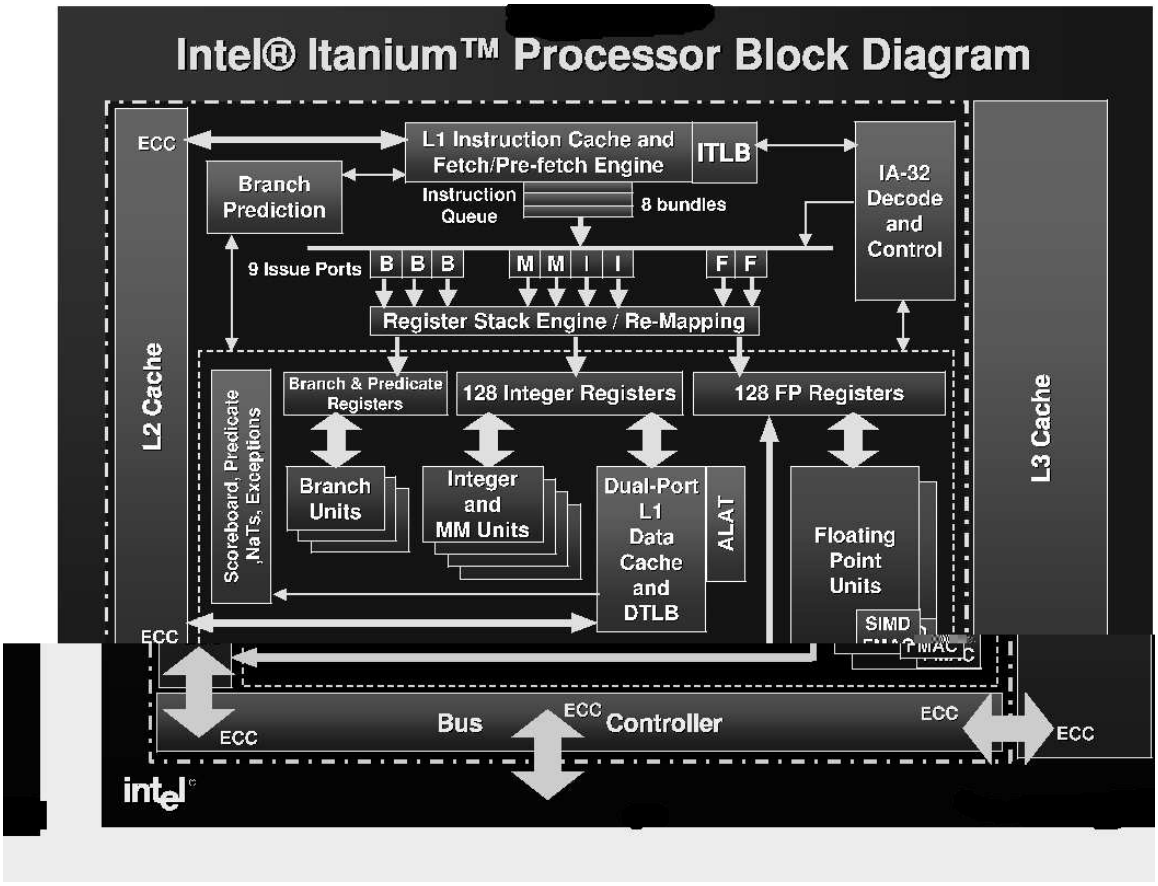


Figure 2.7: Itanium processor core[7]

2.2.7 Crusoe^{TM} Processor

Transmeta Corporation introduced the Crusoe^{TM} processors [73], an x86-compatible family of solution that combines strong performance with remarkably low power consumption. The new technology is fundamentally software based: the power savings come from replacing large numbers of transistors with software.

The Crusoe processor solution consists of a hardware engine logically surrounded by a software layer as shown in figure 2.8. The engine is a VLIW CPU capable of executing up to four operations in each clock cycle. The VLIW's native instruction set bears no resemblance to the x86 instruction set; it has been designed purely

[7]< Source: Sharangpani, "Intel ITANIUM^{TM} Processor Micro-architecture Overview", Intel Microprocessor Forum, Oct 1999 [72] >

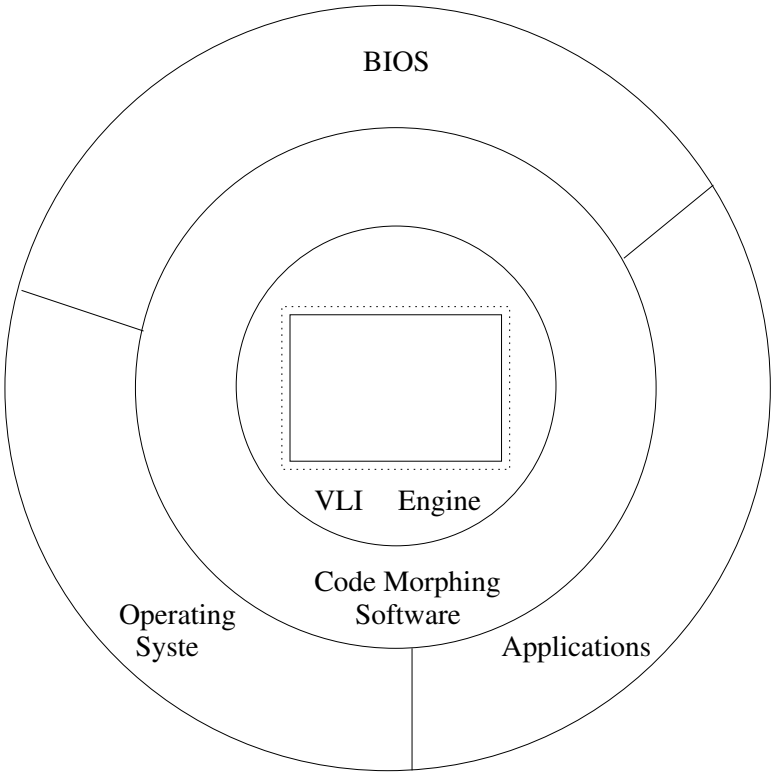


Figure 2.8: Code morphing and the Crusoe processor[8]

for fast low-power implementation using conventional CMOS fabrication. The surrounding software layer gives x86 programs the impression that they are running on x86 hardware. The software layer is called Code Morphing^{TM} software because it dynamically "morphs" x86 instruction into VLIW instructions.

The Code Morphing software is fundamentally a dynamic translation system, a program that compiles instructions for one instruction set architecture (in this case, the x86 target ISA) into instructions for another ISA (the VLIW host ISA). The Code Morphing software resides in a ROM and is the first program to start executing when the processor boots.

Transmeta's Code Morphing technology change the entire approach to designing microprocessors in which practical microprocessors can be implemented as hardware-software hybrids.

[8]< Source: Klaiber, "The Technology Behind CRUSOE^{TM} Processors", Transmeta Corporation, Jan 2000 [73] >

Chapter 3

A Case for Memory to Memory Architecture

In the systems today, the processor-memory speed-gap is a major performance bottleneck and is likely to continue to be a worrying issue in future also. In this chapter, we critically view the two widely used ways of reducing the processor-memory bandwidth gap, namely the registers and cache memory. We also bring out the drawbacks of register-to-register architecture. We then propose a new way of organizing local memory called on-chip registering of memory locations. In this context, we also discuss the advantages of memory-to-memory architecture.

3.1 Background

Predominantly the memory in the systems today is realized using Dynamic Random Access Memory (DRAM). Unfortunately, the speed of DRAM is not improving at the same rate as that of processor (figure 3.1). However, DRAMs are still in use because, they offer the best price/performance ratio due to their extremely low cost and high densities [74, 75].

Programs are written in "infinite" memory model, ever since the introduction of the virtual memory concept. During program execution, both instructions and data are in main memory. Once the program starts executing, the processor fetches the

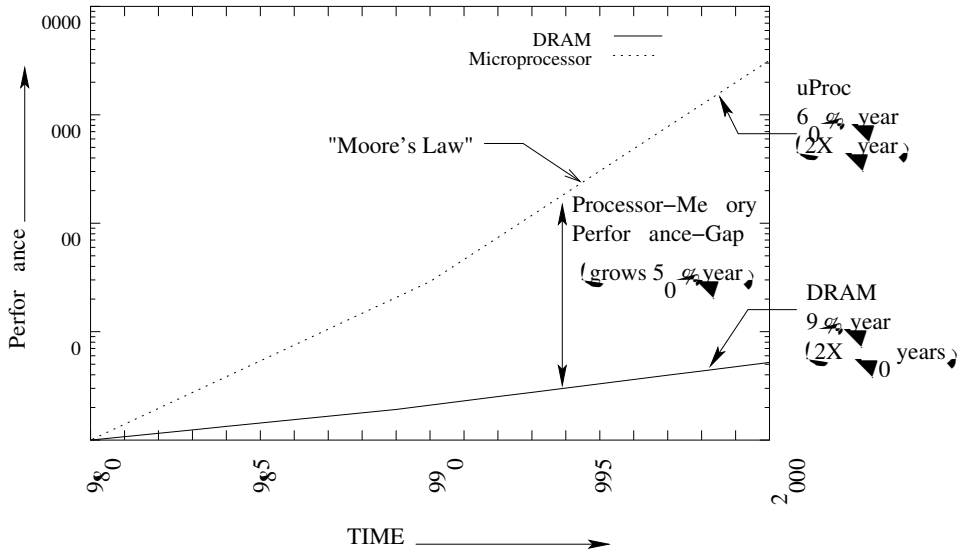


Figure 3.1: Growing processor-memory speed gap

required instruction and data on demand from the memory, using virtual addresses. The execution of the instruction is typically much faster than the fetch of instruction and data from memory. As every operation has to fetch instruction and data from memory, the processor-memory speed-gap is crucial to performance.

Several solutions are used to reduce the growing processor-memory speed gap. Two traditional methods that are widely used even today are on-chip registers and caches. They are primarily a small number of high speed buffers placed between memory and processor to hold the most frequently used data and instructions. In the early days of microprocessors, caches were outside the CPU chip due to technological limits. As VLSI technology improved these were placed inside the CPU. Processors today have large register sets and multi-level caches right inside the chip. Both on-chip registers and on-chip caches are implemented using the same technology as that of the processor. A typical memory hierarchy in systems today is as shown in figure 3.2. Typically, a low cost technology is used for larger memories but it also yields slower access times. Keeping the necessary instructions and data inside the processor (caches, registers) before they are actually required, reduces or eliminates the speed-gap between processor and memory. There are many different ways in

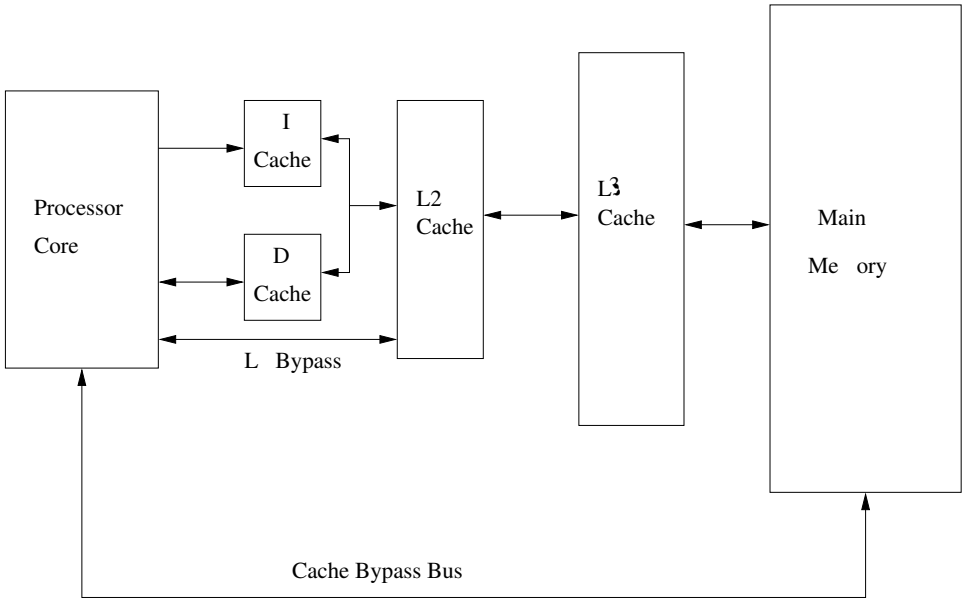


Figure 3.2: Memory hierarchy in modern computers

which registers and caches are implemented and also the ways in which they are used by the programs. Sites in his paper [53] gives an excellent insight on this issue.

3.2 Cache Memory

One of the time-tested mechanisms for improving memory system performance is the use of cache – a redundant and dynamically varying subset of the memory system. Transparent to the user, caches try to hold the most frequently accessed portions of main memory by the program. Cache memory reduces the effective memory access time and main memory traffic by exploiting temporal and spatial locality of reference in program and data. Processors can fetch the required instruction and data much faster if they are in cache (hit) than when they are not in the cache. The caches also reduce the demand on the memory bandwidth by reducing the memory traffic. Caches are typically independent of the architecture and consistently work well by exploiting dynamic program behavior. The caches are invisible to the programmer. An exhaustive survey on cache memories can be found in a paper by Smith [76].

Cache memories, their design and techniques to improve their performance have been a major topic for research over the last two decades. Performance tradeoffs in cache design with respect to the interaction of cache design parameters like *cache size v/s CPU/cache cycle time, set associativity v/s cycle time and block size v/s main memory speed* is well addressed by Przybylski et al. [77]. Performance of on-chip cache memories with *miss ratio* and *traffic ratio* as metrics is presented by Hill and Smith [78]. Several different local memory organizations applicable for single-chip processors like *instruction cache, data cache (split and unified), stack and multiple register set* are described and studied in a paper by Eickmeyer and Patel [79].

Current day processors have extremely powerful caching mechanisms. All of them have large on-chip split instruction and data cache as shown in figure 3.2. The effective access time of on-chip caches with some architectural enhancement features comes close to that of the registers. The access time of cache is comparatively slow when compared to that of registers, because they are addressed using memory addresses, which are long. Further, as the cache size increases the access time also increases because of increased complexity in the decoding logic. However, various tasks associated with cache access, such as TLB (Translation Lookaside Buffer) access, tag comparison and actual cache access are pipelined making their effective access time quite comparable to that of registers. Many of today's processors also support simultaneous multiple access to the cache [80, 15, 12]. They either duplicate the entire cache [26] or use interleaved memory banks [15, 17] to implement multi-port cache. They employ lockup-free techniques wherein the cache system continues to receive memory requests from the process, even when a miss occurs (non-blocking cache). Some of them employ techniques like *miss caching* and *victim caching* to reduce conflict and capacity misses [80]. Some of them employ techniques like *line buffer* and *stream buffers* to reduce compulsory misses. Load-all technique improves the port efficiency of caches by serving multiple reads to the same word simultaneously. Load-all-wide builds on load-all by serving multiple reads to the same cache block [58].

Superscalar processors place a very high demand on instruction bandwidth. To meet this demand all modern processors have efficient fetch mechanisms supported

by large on-chip instruction caches, wide buses and branch prediction. Instruction bandwidth today is not viewed as a bottleneck. However, the same thing cannot be said with data. It has been seen by experience, that most programs will give instruction hit ratios of above 90% even with a small cache of 1 KB [81]. Many processors today also have large L2 caches and controllers for L3 cache on the chip [19, 15]. In the current technology, processors have about 8–64KB of on-chip L1 instruction cache and about 8–64KB dual port L1 data cache. The entire process of cache access is pipelined to give an effective access time of one or two cycles.

3.3 Registers

Registers are used for various reasons. Their small size and proximity to the ALU offers faster access speeds than that of the main memory. Registers are small in number resulting in short addresses and less complex hardware for decoding them than the one for addressing cache or main memory. They provide a physically separate memory that can be accessed in parallel with the main memory. While caches depend on the dynamic program behavior to get the data and instructions into them, registers depend entirely on compilers and programmers to explicitly load the required data into them. Compilers and programmers use exclusive load/store instructions to move data between memory and registers. Once the operands are loaded into registers from memory they can be accessed at very high speeds until the same register is used to hold a different data. The registers therefore improve the effective memory access time and reduce the pressure on the memory bandwidth. The registers provide multiple simultaneous accesses such that one can perform several reads and one write operations on a register in the same clock. Since their accesses do not clash with the memory access, compilers can reschedule the instructions for better performance. Because of their short addresses, performing data dependence analysis and breaking the false dependencies using dynamic register renaming is much simpler than doing the same with the cache memory.

Compilers perform global register allocation to determine which values to keep in registers. The superscalar processors on the other hand perform dynamic instruction

scheduling by looking through a window of instructions in the instruction stream. However, compilers also reorders instructions to avoid pipeline hazards. These three techniques are typically performed independent of each other. In reality, these steps are dependent on each other and can impose constraints on the other, sometimes producing inefficient code [21]. Keeping a large number of values in a small number of registers creates a large number of conflicts when the execution order is changed from the order assumed by the register allocator [82].

Code inefficiencies creep in when a memory location is accessed or when pipeline delays occur. If the register allocation is performed first, the same physical registers may be assigned to hold values from independent expressions, reducing the potential ILP. This, in turn, decreases the scheduler's opportunities to mask pipeline delays. If the scheduling precedes register allocation, the number of simultaneous live values may be increased, creating register spills to memory. Many studies have focused on issues related to register allocation in the context of dynamic scheduling [82, 83, 84]. A novel method called RASE (Register Allocation with Schedule Estimates) [85] integrates register allocation and instruction scheduling by giving register allocator cost estimates that quantify the effect of its allocation choices on the subsequently generated schedule. Several other works also address this problem [39]. In summary we can say that register induce *artificial dependence* among program variables which were not there in the original program.

In register-to-register machines, all the operands have to be in registers before any operation can be performed on them. Load/Store instructions account for approximately 30-40% of dynamically executed instructions [22]. Further, registers have to be saved across procedure calls. As registers are not typed and memory are, additional instructions are required to type convert data in the registers with respect to the data type of the values they hold. An excellent study is done by Franklin and Sohi [86] about register traffic analysis in a load-store architecture. In this paper the authors have found that more than 95% of register instances are used at most 3 times. Those instances which are used more than 4 times constitute primarily the read traffic. About 10% of register instances constitute 20-30% of source operands of all instructions. They also find that holding a few register instances in a buffer

in execution unit can reduce read register traffic by 80% and write traffic by 50%.

All processors today have a separate set of integer and floating point register sets. Register sets in current superscalar processors are very complex with large number of ports typically in the range of 8–10 read ports and 4–6 write ports for a four issue processor. Manufacturers use lot of tricks to implement this, Alpha 21364 duplicated the integer register file, with each copy having 4 read ports and six write ports, giving a total of 8 read ports and 6 write ports (Writes are performed on both the copies simultaneously) [19]. In contrast, the register file in Alpha 21464 has 16 read ports and 8 write ports. In superscalar processors, much of the data flow activity is concentrated around the rename registers and reorder buffer. Registers are renamed dynamically and placed in the rename registers. The subsequent instructions read the register from the rename registers rather than the register set itself.

In this scenario our argument is that the on-chip registers as used today in the memory hierarchy is not the best way of using for the following reasons.

- Programs are written using infinite memory model.
- Compilers are too conservative and fail to find variables to allocate registers.
- Very few register instances (about 5%) are used more than three times.
- While compiling programs to superscalar processors, compilers fail to generate efficient code and often introduce data dependences that were not there in the original programs.
- Caches today offer effective memory access times which are comparable to that of registers.
- Additional instructions are necessary to adjust the data in the registers according to the data type that it had when loaded from the memory or when stored in the memory.
- The operands that are brought into registers by explicit load instructions are also brought into cache at the same time and hence could be accessed at the speed of the cache.

- Explicit Load/Store contributes about 30% of the instructions in a register-to-register architecture.
- Registers contribute large overheads during procedure calls.

We argue that the same multi-ported register set can be used as say a level zero cache (L0). In other words, the first level of memory hierarchy is realized using registers. We believe due to the program dynamics these small number of memory locations will be able to automatically capture bulk of the frequently used operands. This coupled with a pure memory-to-memory instruction set will eliminate many of the overheads associated with the register set.

3.4 On-chip Registering of Memory Locations

We call this concept of L0 cache (figure 3.3) the on-chip registering of memory locations. This concept serves well to implement a pure memory-to-memory architecture. This renaming of memory locations to on-chip registers is transparent to the programmer and the performance purely depends on locality exhibited by the programs.

As the registers are mapped on to memory locations, they have long addresses unlike conventional registers, which have short addresses. Since all programs exhibit locality, the on-chip registers are expected to observe very high hit ratio. If properly supported by large on-chip caches, the miss penalty on a register miss will be very low and can be kept to a single clock if it hits in L1 and to a few cycles if it hits in L2.

With the modified scheme of registers, compilers can do a better job in exploiting available ILP as they need not perform register allocation. They need to optimize only on scheduling the instructions. As a large number of temporary variables are not necessary in a memory-to-memory architecture, the on-chip registers can capture more useful locality. By exposing cache to programmers, compilers can optimize the code to prefetch data into cache.

Further, we can place a small set of high speed buffers in the execution path and dynamically schedule the instructions. The instructions waiting for the operands will

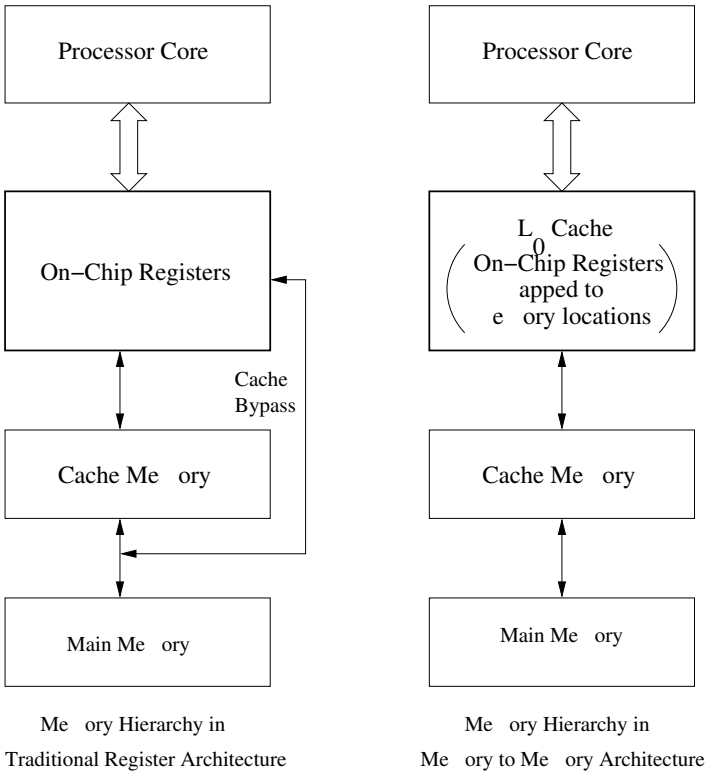


Figure 3.3: On-chip registering of memory locations

get them either from cache or from this buffer (data forwarding). If the operands are still in the process of computation, they are dynamically renamed to the physical buffer assigned to hold the required value. Thereby a superscalar implementation of a pure memory-to-memory architecture is possible. Even then, one has to find better ways to handle the long addresses of memory while analyzing data dependences.

3.5 Memory to Memory Architecture

The memory-to-memory processors perform operations directly on the memory address space.

Programs are compiled with a particular memory model in consideration. High

level programs typically use variables in the memory. Thus, a pure memory-to-memory instruction set has the highest capability to capture the program behavior as written by the programmer. Therefore, the compiled programs use the least number of instructions in the memory-to-memory instruction set. The expression for execution time (eqn. 3.1), gives a powerful formulation tool for analyzing, comparing and projecting processor performance.

$$\frac{\textbf{time}}{\textbf{program}} = \frac{\textbf{instructions}}{\textbf{program}} \cdot \frac{\textbf{cycles}}{\textbf{instructions}} \cdot \frac{\textbf{time}}{\textbf{cycle}} \tag{3.1}$$

or

$$\textbf{T} = \textbf{N} \cdot \textbf{CPI} \cdot$$

Programs compiled for memory-to-memory processor exhibits the most compact code, hence the first term (N) in equation 3.1 is low when compared to a register-to-register architecture. With careful design of instruction set and efficient hardware implementation the average number of cycles per instruction can be kept low. By using the best available technology and efficient pipelines the machine cycle time ( ) can be kept low. The characteristics of register-to-register architecture have a natural ability to keep the CPI and low. However, with the speed of cache memory reaching close to that of the registers, the same thing can be worked out for memory-to-memory processor.

The presence of large and fast caches inside the CPUs and the constant improvements in VLSI technology provide several choices for an efficient memory hierarchy. Current processors can perform simultaneous multiple cache accesses in a single cycle.

Traditionally memory-to-memory processors were not considered as ideal choices for pipelining. But with the presence of large, fast caches which are as fast as registers and with an orthogonal instruction set with a few simple addressing techniques, we believe a pipelined implementation of memory-to-memory architecture is not only possible but is more efficient than the register architecture.

3.6 Analysis

In our work, we named the register–less (read register address space less) architecture as Performance Enhanced Register–Less (PERL) architecture. We compare the PERL processor with the DLX, a hypothetical processor presented by Hennessy and Patterson [22]. We chose DLX, because it is a classic example for a register-to-register architecture. Further, it is the most chosen architecture to study many of the architectural novelties, especially in the academia. Excellent simulation tools are available for DLX across the World Wide Web. Many statistics describing the performance aspects of DLX are well documented in [22].

We take the dynamic instruction count statistics of DLX as provided by Hennessy and Patterson [22]. We analyze the performance of PERL processor against the DLX using an analytical model. For the analysis, we make the following assumptions.

1. As the PERL processor operates directly on operands in memory, there are no explicit Load/Store instructions. The Load/Stores amounts to 30-40% of dynamically executed instructions (we will average it to 33%) in the DLX applications. Even though there are fewer instructions in PERL programs, each instruction is long and may not result in saving in the number of bytes required to store the program.

2. The number of other ALU instructions executed is the same in both DLX and PERL. Though PERL is likely to execute less number of instructions as extra instructions to adjust the operands in registers are not required.

3. We also assume that all the operands in an ALU instruction of PERL use indirect mode address and hence require 5 reads and one write to memory. That is we assume worst case always, however, in reality many operands will be accessed using direct and immediate addressing modes.

4. The number of dynamically executed branches will be same in PERL and DLX. Further, we take the cost of branch as one clock in DLX, assuming the branch prediction performs with great accuracy. We assume all the branches

in PERL are indirect and hence need two memory access (always the worst case).

The above assumptions are carefully made so that there is absolutely no benefit given to PERL. In fact, we have assumed worst case memory requirement for PERL. From the data provided in [22], we extract that the instruction-use break up of the DLX is 25-45% of Load/Store, about 45-65% of ALU and about 7-20% of branches. For our analysis, we assume that there are about 33% of Load/Store, 54% of ALU and 13% of branch instructions.

The computation of effective access time for n simultaneous accesses in an n-ported cache is computed by modeling n accesses as a sequence of Bernoulli trials [87].

The outcome of each access of the n accesses per cycle is either a hit or miss (mutually exclusive) and are independent of each other. The outcome of each trial can be represented by an n-tuple of 0's and 1's as follows.

$$\mathbf{O}_n = (\mathbf{R}_1; \mathbf{R}_2; \cdots; \mathbf{R}_n) \ \textbf{Where } \mathbf{R}_i \begin{cases} 0 & \text{if the access on } \mathbf{i}^{th} \text{ port results in a miss} \\ 1 & \text{if the access on } \mathbf{i}^{th} \text{ port results in a hit} \end{cases}$$

The sample space of the outcome of the above trial is defined by

$$\mathbf{S}_n = \{2^n \ \mathbf{n-tuples\ of}\ 0'\mathbf{s\ and}\ 1'\mathbf{s}\}$$

The probability assignment over the sample space $\mathbf{S}_1$ is already specified:

$$\begin{aligned} \mathbf{P}(1) &= (\mathbf{h}) \text{ hit ratio in single port cache} \\ \mathbf{P}(0) &= (1-\mathbf{h}) \text{ miss ratio in single port cache} \end{aligned}$$

The probability of any sample point of $\mathbf{r}$ hits and $(\mathbf{n}-\mathbf{r})$ misses is assigned a probability $\mathbf{h}^r \cdot (1-\mathbf{h})^{(n-r)}$. Noting that there are ${}^n\mathbf{C}_r$ such sample points, the probability of obtaining exactly $\mathbf{r}$ hits in $\mathbf{n}$ accesses is given by

$$\mathbf{p}(\mathbf{r}) = {}^n\mathbf{C}_r \cdot \mathbf{h}^r \cdot (1-\mathbf{h})^{n-r} \tag{3.2}$$

It can be verified that equation 3.2 is a legitimate probability assignment over the sample space $\mathbf{S}_n$ since:

$$\begin{aligned} \sum_{r=0}^{n} {}^{n}\mathbf{C}_r \cdot \mathbf{h}^r \cdot (1-\mathbf{h})^{n-r} &= (\mathbf{h} + (1-\mathbf{h}))^n \\ &= 1 \end{aligned} \tag{3.3}$$

The average access time for n accesses in an n-ported cache is given by equation 3.4, where $\mathbf{h}$ is the hit ratio, $\mathbf{t}_{hit}$ is cache access time and $\mathbf{m}_p$ is the miss penalty. Every access that results in a miss is served in $\mathbf{m}_p$ cycle (i.e, no overlap), where as up to n hits can be served in $\mathbf{t}_{hit}$ cycles.

$$\mathrm{T} = \sum_{r=0}^{n} {}^{n}\mathbf{C}_r \cdot \mathbf{h}^r \cdot (1-\mathbf{h})^{n-r} \cdot (\mathbf{r} \cdot \mathbf{t}_{hit} + (\mathbf{n}-\mathbf{r}) \cdot \mathbf{m}_p) \tag{3.4}$$

$$\textbf{Where } \mathbf{r} \cdot \mathbf{t}_{hit} = \begin{cases} \mathbf{t}_{hit} & \textbf{if } 1 \leq \mathbf{r} \leq \mathbf{n} \\ 0 & \text{if r} = 0 \end{cases}$$

We take the equations for Clocks Per Instruction (CPI) as the base for reference. In the equation for CPI_{DLX} (eqn. 3.5), 54% instructions take only one clock to execute (ALU), 33% instructions (load/store) perform one memory access and hence depend on the hit ratio ($\mathbf{h}$) and miss penalty ($\mathbf{m}_p$), 13% instructions (branch) are assumed to execute in one clock.

In the equation for CPI_{PERL} (eqn. 3.6), 54% instructions (ALU) are assumed to perform six memory accesses (worst case) and hence depend on hit ratio ($\mathbf{h}$) and miss penalty ($\mathbf{m}_p$), the 13% (branch) instructions are assumed to perform two memory accesses (again the worst case) and hence depend on $\mathbf{h}$ and $\mathbf{m}_p$. 33% (load/store) instructions are not applicable for the PERL. We assume that the L0 cache has 6 ports and has an access time of 1 cycle. However, there are no explicit load/store instructions. Hence the equation for $\mathbf{CPI}_{PERL}$ is normalized with respect to the instruction count of DLX, where as equation 3.5 gives the actual CPI for DLX.

$$\mathrm{CPI_{DLX}} = 0{:}54 + 0{:}33 \cdot \mathbf{h} + 0{:}33 \cdot (1-\mathbf{h}) \cdot \mathbf{m}_p + 0{:}13 \tag{3.5}$$

$$\begin{aligned} \mathrm{CPI_{PERL}} = {} & 0{:}54 \cdot \sum_{r=0}^{6} {}^{6}\mathbf{C}_r \cdot \mathbf{h}^r \cdot (1-\mathbf{h})^{6-r} \cdot (1 + (6-\mathbf{r}) \cdot \mathbf{m}_p) + \\ & 0{:}13 \cdot \sum_{r=0}^{2} {}^{2}\mathbf{C}_r \cdot \mathbf{h}^r \, (1-\mathbf{h})^{2-r} \cdot (1 + (2-\mathbf{r}) \cdot \mathbf{m}_p) \end{aligned} \tag{3.6}$$

Using normalized CPIs we can calculate the CPU execution time of the program on both the machines given only the total number of dynamically executed instructions on DLX($\mathbf{N}_{DLX}$) as shown in equations 3.7 and 3.8

$$\textbf{CPU Execution Time on DLX} = \mathbf{N}_{DLX} \cdot \mathbf{CPI}_{DLX} \quad (3.7)$$

$$\textbf{CPU Execution Time on PERL} = \mathbf{N}_{DLX} \cdot \mathbf{CPI}_{PERL} \quad (3.8)$$

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 3.4 and 3.5. Figure 3.4 gives the variation of normalized CPI for both DLX and PERL for varying hit ratio. The curves are just the plots of equations 3.5 and 3.6 with constant miss penalty ($\mathbf{m}_p$) and variable hit ratios (0.95–1.0).

Equations 3.7 and 3.8, show that the performance of DLX and PERL are equal when the normalized CPI of PERL and DLX are equal. So, if we equate equations 3.5 and 3.6 and solve for miss penalty ($\mathbf{m}_p$), then we get an equation in terms of hit ratio ($\mathbf{h}$). If we substitute a value for $\mathbf{h}$ in the resultant equation, we get the miss penalty at which both DLX and PERL will have same CPI, this we call as tolerable miss penalty. Figure 3.5 shows the tolerable miss penalty for hit ratios varying from 0.9 to 0.99.

The figure 3.4 clearly shows that at higher hit ratios PERL exhibits a lower normalized CPI and hence would perform better than DLX with reference to execution time. The maximum CPI that the DLX can achieve is 1, while PERL achieves CPI (normalized one) of less than one at higher hit ratios ($\mathbf{h} > 0{:}99$). However, it may be noted that compilers can further reduce the CPIs in both the cases. 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.

The plots in figure 3.5 clearly shows that at higher ratios both DLX and PERL can tolerate higher miss penalties. Typically, in processors today it requires about 6–8 cycles to access the L2 cache on a L1 cache miss. From the plot in figure 3.5 we can

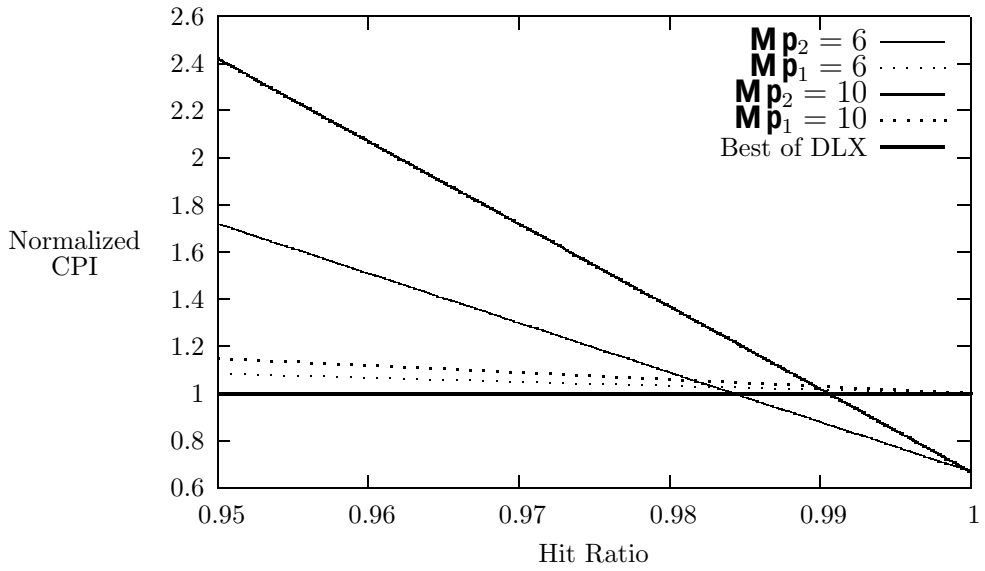


Figure 3.4: Normalized CPI vs. hit ratio

see that PERL has to achieve a hit ratio of 0.985 or more to perform as good as DLX. The plot does not look very appealing for PERL as the plot indicate that PERL would perform better only when miss penalties are very low. However, in practice PERL has a better hit to the cache than the DLX as all the register accesses in DLX (traditional register machine) are turned into memory (cache) access in PERL. The register access of DLX therefore translates to cache hit in PERL. Intuitively we can expect the number of misses will be the same in both DLX and PERL. As PERL has more memory accesses than DLX, the hit ratio for PERL are much better than the DLX. Therefore, we can expect PERL to perform better than DLX, especially at high hit ratios. With a perfect cache PERL will definitely outperform DLX as shown in figure 3.4

In future, processors can be built with large on-chip memory enough to hold the entire program and data in them [65]. Compilers would be able to further boost the performance of PERL by performing proper code scheduling, prefetching and producing valuable hints in operand access and branch prediction. This along with the expected high performance of PERL at higher hit ratios gives us enough motivation to further investigate the case of register–less processor.

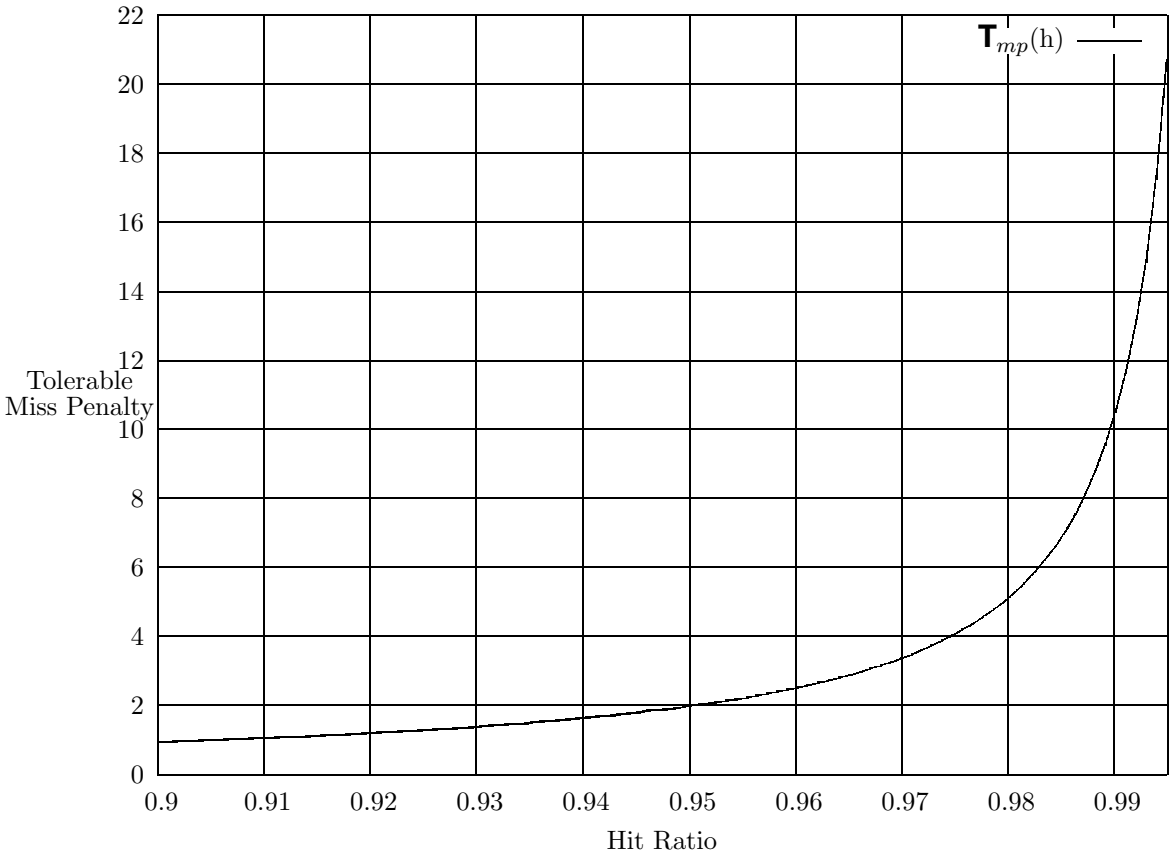


Figure 3.5: Tolerable miss penalty vs. hit ratio

Chapter 4

PERL - A Memory to Memory Architecture

In this chapter, we present an example design of a memory-to-memory processor. This processor is defined for the study purposes and we do not claim it to be the ideal one or the best. Many alternatives to the architecture are possible and here we define and explore one of them.

As discussed earlier, the PERL (Performance Enhanced Register–Less Processor) has the following features.

- It has no explicit register name space.
- The first level of memory hierarchy L0, consists of a set of high speed registers mapped to memory locations in a manner similar to mapping the cache to memory. The organization of L0 hierarchy is implementation dependent and is invisible to the programmer.
- All instructions operate on memory and hence there are no explicit load/store instructions.
- The instruction set is conducive for superscalar execution.

4.1 Instruction Set Architecture

PERL instruction set is based on a pure memory-to-memory architecture. We use three address format that provides maximum code compaction [22]. All instructions operate on memory and have a general format as follows.

`instruction_mnemonic dest<:dType>, src1<:dType>, src2<:dType>`

for example, `add _p:b1,_q:b2,_r:b4`

Here dest, src1 and src2 are all memory addresses. The first operand dest is the destination of the operation on memory locations src1 and src2. Data type of each operand can be specified with the associated dType fields. For example in the instruction above, `add` is performed on data stored in location _q and _r. The sizes of the operands are two bytes and four bytes respectively. The result is stored in 1 byte location at _p. A shorter version of the instruction when all operands and destination have same data type can be written with dType associated with the instruction_mnemonic. For example the instruction `addb4 _p,_q,_r` is a short form of the instruction `add _p:b4,_q:b4,_r:b4`.

Details of the PERL instruction set are provided in Appendix A.

4.1.1 Addressing Modes

Addressing modes have the ability to significantly reduce instruction counts (N in equation 3.1). They however, also add complexity in the micro-architecture of the processor and increase the average number of cycles per instruction (CPI). A judicious choice of them can keep the instruction count (N) low without such complexities. Several studies have been done related to instruction sets and their usage [88, 89, 90, 91, 92, 9].

In choosing the addressing techniques for PERL, we consider the results reported by Hennessy and Paterson [22]. They made the following observations about the characteristics with reference to the addressing techniques used by the compilers, while generating memory addresses (i.e., for memory operands in ALU and Control instructions and memory addresses in Load/Store instructions). They use VAX

architecture for measurements related to memory addressing and MIPS/DLX architecture for measurements on the usage of instruction sets of current machines.

- Frequently used memory addressing modes are displacement, immediate and register indirect and these represent 75-99% of all the memory addressing modes used in a program.
- A large percentage (75%) of displacement values could be represented within 12 bits. This percentages increase to 99% if 16-bit offsets are used.
- About 50% to 70% of the immediate constants can be accommodated using 8 bits. This number increases to 75% to 80% if 16 bits are available for storing constants.
- Memory indirect addressing represents only a very small percentage (1-6%) of all addressing modes used.
- Offsets for the PC-relative jump instructions can be accommodated within 8 bits for most of the instructions.
- In programs for the VAX architecture, register addressing modes account for one half of the operand references, while memory addressing modes (including immediate) account for the other half.

As a consequence, load/store instructions in all modern instruction sets have register indirect with immediate index addressing mode (EA = register + immediate). Some processors also support register indirect mode (EA = register) and register indirect mode with register index (EA = register + register). This results in a reduction of instructions per program in certain applications (to the order of 5–6%) [92].

4.1.2 Key Features

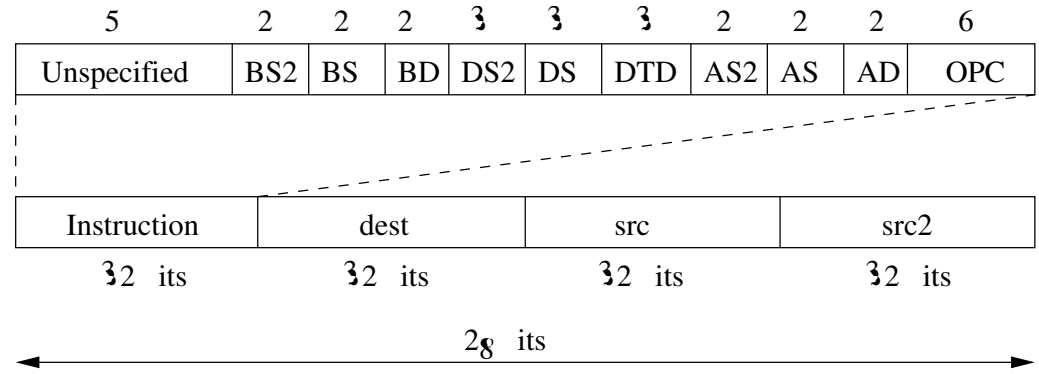
PERL instruction set has the following features based on the studies as described previously and the fact that it is a memory-to-memory architecture.

1. All instructions are of the same length and use three address format.

2. The machine supports eight integer data types namely 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.

3. The two operands are specified by any one of the four addressing methods namely direct, indirect, displacement and Immediate. The destination of the instruction is also specified by any of the four addressing modes. If an immediate addressing mode is used for the destination, the result of the instruction is not saved and this instruction then almost behaves as NOP. The source operands are however fetched from memory (and therefore may result in page faults). Compilers can use such an addressing mechanism to prefetch data into caches.

   In displacement addressing mode the instruction should specify a base address and displacement. To specify the base address, PERL provides a Third



OPC	Operation Code
AD, AS & AS2	Addressing odes for destination and source operands
DTD, DS & DS2	Data types of destination and source operands
BD, BS & BS2	Base pointers for destination and source operands

Figure 4.1: PERL instruction encoding

```
mul _p:b4, _q:b2, _r:b2
divf8 _i, _j, _k
```

6. Jump and conditional branch instructions are also supported. Call to functions and return from them are implemented as jumps. In conditional branch class of instructions the dest field specifies the target address, the src1 and src2 addresses specify the operands used to evaluate the condition. For example the instruction `jltb4 L19, -16(fp), _firstsquare` represents a conditional branch which is taken when `-16(fp)` is less than `_firstsquare`.

   In the unconditional jump instructions, the dest field specifies the target address, src1 field, if used, specifies the address where the current PC value will be saved. The src2 field of the instruction is not used. Following example shows the way in which call and return could be realized using jump instructions,

```
j _init, -8(sp) ;; call _init by saving PC onto stack
j -8(sp) ;; return
```

Among the four memory locations cached permanently in the L0 cache, *gcc* compiler uses two for stack pointer (SP) and frame pointer (FP). Various other compilers can use these locations for storing intermediate values (the same way as they use registers in a conventional processor) or array bases for base addressing.

It can be observed that the instruction length in PERL is very long. PERL instructions load upto two operands, and operate on them before storing one result. In an instruction, if both the operands and the destination are specified using indirect addressing, the processor has to perform as many as 6 memory accesses to execute such an instruction. As one would expect, the performance of PERL is highly dependent on the cache hit rates. However, some instructions in a conventional processor where registers are used to keep values, will corresponds to the PERL instructions that access memory for operands or no instructions at all. Thus, the program locality is higher compared to the conventional processor and typically results in a better cache hit ratio.

There are some additional advantages of the PERL processor compared to a conventional processor. In Reg–Reg machine all operations where operands are less than the size of the register require additional instructions for data conversions such as sign-extension, bit-masking etc. In PERL processor, however all operands are in memory. The types of the operands and destination are specified in the instruction itself. This does not require additional instructions for doing so. The overhead in context switch is also minimal as the machine state is very small (only PC, SP and FP).

4.2 Processor Datapath

PERL processor executes the instructions in a pipelined fashion. A pipelined execution of instructions benefits both the superscalar and non-superscalar implementation of the processor. The datapath of PERL processor pipeline is given in figure 4.2.

4.2.1 Pipeline Stages

The basic pipeline comprises of fetch, decode and address generation, operand access, execute and write back stages as shown in figure 4.2.

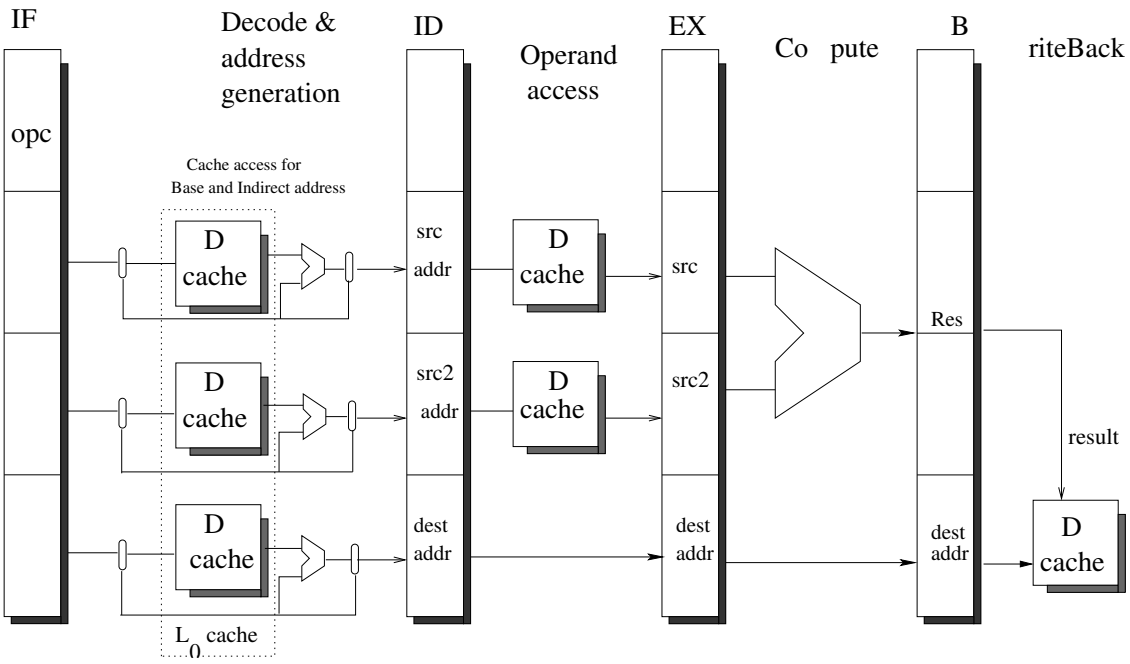


Figure 4.2: The processor datapath of PERL

Fetch stage: This stage brings in the next instruction from the I-cache and places it in the instruction queue for decode. In branch instructions, the next instruction fetch depends on the outcome of branch execution. The branch hazards are reduced using branch prediction mechanisms and are explained later in section 4.4.1.

Decode and Address generation stage: In this stage, the instruction that was fetched in the previous cycle is taken from the buffer and decoded. The effective addresses of dest, src1 and src2 of the instruction are computed based on the addressing modes and placed in the buffers. If the addressing mode is direct or immediate then the memory is not accessed to get the address of the operand. In such cases, address/value present in the instruction is forwarded to the buffers of the next stage. If an operand is addressed using 2-bit base or indirect addressing modes, a memory access is performed to get the base/address. This access always results in a hit in

L0 cache which contains the base pointers cached permanently. In all such cases data dependencies on the other instructions in the pipeline are resolved before the memory access is performed. An ALU operation is required for base addressing mode, to add the offset specified in the instruction with the base value. Thus, this stage performs upto a maximum of three memory accesses (to L0 cache) and upto a maximum of three additions.

Operand access stage: This stage uses the effective address computed in the decode stage to read the operands from the memory. If required operand is not found in L0 or L1 cache then a memory access is performed and the pipeline is blocked until the data is brought into the cache. Data dependencies are also resolved before the memory access.

Execute stage: In this stage, the actual operation on the operands is carried out. In case of a branch instruction, outcome of the branch instruction is known at this stage. If the prediction was wrong, the instructions following the branch are *flushed* and the BTB is updated. The computed result is placed in the buffer of the next stage and also forwarded to the earlier stages if required.

Write Back stage: The write back stage writes the result computed into the memory location pointed by the destination address.

Each PERL instruction execution results in 1 fetch (I-cache), upto 5 data reads and 0 or 1 write access to the cache. Thus, the cache system design is extremely important for the efficiency of the processor. In the worst case when all memory accesses result in a cache miss, PERL can produce upto 5 memory reads and one memory write. By restricting the base addresses to some fixed locations in the memory and registering them on chip (L0 cache) brings down the maximum reads from 5 to 2. Cache port efficiency improvement techniques like Read All and Read All Wide [58] also benefit PERL to a great extent (as discussed later in section 4.5.1).

4.3 Superscalar Processor Model

Superscalar execution of instructions is an attractive technique to exploit the ILP in a program. Hence, it is necessary for us to discuss how a superscalar model of

PERL is implemented. Techniques like branch prediction, out-of-order instruction issue, renaming etc., are employed to extract ILP from the programs. Superscalar processors need multiple functional units so that they can issue multiple instructions in the same cycle.

PERL simulator is highly configurable and the configuration of a superscalar version can be specified with varied parameters. A particular superscalar processor model of PERL is shown in figure 4.3, that has two operational units – an integer unit and a floating point unit. These operational units are supplied with instructions 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 techniques embedded into PERL architecture. These techniques are not specific to the superscalar model but can be used to improve the performance of non-superscalar pipeline as well. However, their benefits are generally more pronounced in superscalar processors.

- **Fetch**. The instruction fetcher brings in a fixed number of successive instructions in program order from instruction cache and places them in an instruction queue. Branch prediction is used to speculatively fetch instructions from the predicted target address.

- **Decode**. The decode unit performs its operation in two stages. A fixed number of instructions are taken in program order from the instruction queue, decoded and dispatched to the appropriate central window (Integer/Float). The address of dest, src1 and src2 are computed for each of the instruction decoded in exactly the same way as discussed in section 4.2.1.

  The actual operand access is done in the second stage. Data dependencies are resolved by placing the values of the instructions operands in the window entry. Each operand address is first searched in the reorder buffer to see if the operand's value is generated by a previously issued instruction. If found and its value is not yet valid, then the reorder buffer entry is taken in place of the operand value. If the operand value in the reorder buffer is valid then that value is taken. The reorder buffer is searched starting from the tail and

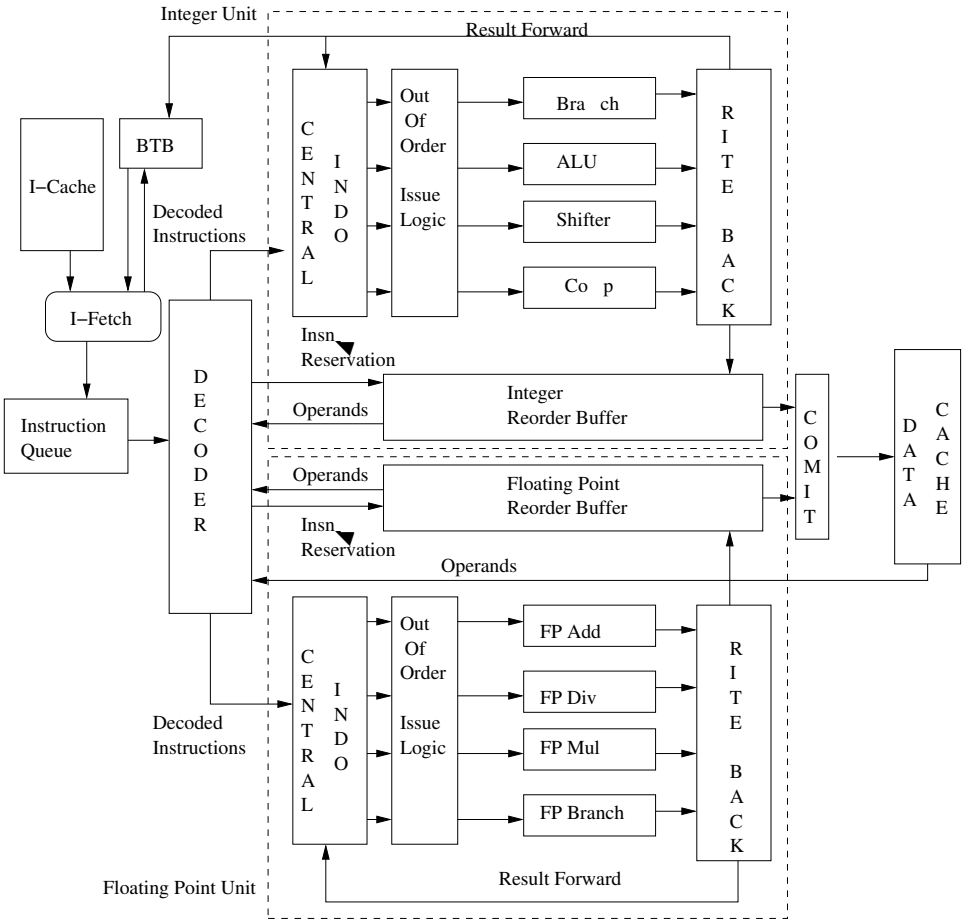


Figure 4.3: The superscalar PERL processor model

the first match is taken as it holds the most recent value of the operand. The operand is fetched from the memory (cache) if it is not found in the reorder buffer. The operands can be fetched out-of-order, i.e. the operands of a later instruction can be fetched before the operands of earlier instruction.

Such a technique decouples instruction decoding from instruction execution thereby simplifying dynamic scheduling. For every instruction that is decoded, an entry is made in the corresponding reorder buffer and its destination address is placed in it. The entries in the reorder buffer are made in program order.

- **Out-of-order instruction issue**. Instruction-issue logic examines instructions in the window, instructions whose operands are available and the required

functional unit is available are termed as *ready* instructions. The issue logic picks up a fixed number of ready instructions, not necessarily in program order, and dispatches them to their appropriate functional unit. When more than one instruction needs the same functional unit, the oldest one among them gets the priority. Thus, there can be many active instructions as long as there are no resource conflicts.

- **Out-of-order completion**. Because of the out-of-order 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.

  All the results of the instructions are written into the reorder buffer by the write back unit. 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.

  The outcome of the branch instruction is compared with the corresponding prediction made. If the prediction turns out to be 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. Results are also forwarded to those instructions in central window waiting for operands.

- **In-order commit**. The validated results are written back to memory (data cache) 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. By this, instructions are made to complete in program order. The committed instructions are removed from the reorder buffer. Invalidated instructions that follow a mis-predicted branch are simply discarded.

4.3.1 Important Resources

Some important resources and mechanisms are used to accelerate instruction processing and thus enhance the above superscalar features. They are the following.

- **Reorder Buffer.** There are two instances of reorder buffer one in each operational unit. They are implemented as circular queues with a head position and tail. Newly decoded instructions are assigned an entry at the tail of queue, and they commit in order at the head. With this mechanism, instruction entries do not have to move towards the bottom or top of the queue, instead only the head and tail pointers are adjusted. Thus, the memory addresses of the results of the instructions get mapped to reorder buffer entries. This may complicate cache coherence protocols if the PERL processor is used in an SMP machine. All the reorder buffer entries are going to commit at appropriate time in their natural course, if some other processor writes into the memory location that is currently in the reorder buffer then the reorder buffer entry has to be invalidated. This approach gets complicated and may bring down the performance if there are many such invalidations. A better approach is to obtain exclusive copy of the location in the reorder buffer thereby blocking all other processor from writing into that location. The exclusive rights are released once the location is written back to the L1 cache.

  The reorder buffer is one of the major source for operands. It holds the results of the instructions, which have completed out-of-order which in turn, may be required as operands for the following instructions. It captures the temporal locality of the data very effectively. The fact that these operands can be supplied from the reorder buffer reduces the accesses to memory.

- **A branch target buffer (BTB)** between the instruction fetch and the decoder enables branch prediction 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 fetch unit starts fetching instructions from the target address in case of a direct jump.

- **A multi-ported interleaved data cache** is provided to support multiple data accesses due to multiple instructions being executed. Techniques to improve the cache bandwidth like victim caches, line buffer, load-all-wide etc., yield extremely good benefits depending on the address pattern. A wide bus between instruction cache and Instruction queue is provided as it has to carry multiple instruction words.

4.4 Branch Prediction

The branch instructions in PERL instruction set are categorized as conditional direct, conditional indirect, unconditional direct and unconditional indirect. It may be noted that conditional indirect branches are used very infrequently [93].

4.4.1 2-bit Branch Prediction

Our initial design of PERL [94] made use of a 2-bit branch prediction as described by Lee and Smith [95]. This scheme improves the performance of both pipelined and superscalar versions of PERL. In this scheme past behaviors of a branch are recorded and used to find the outcome of the branch instruction. For branch prediction, PERL uses a Branch Target Buffer (BTB), whose contents are used by during the instruction fetch to predict the outcome of the fetched branch instruction. Each entry in this BTB contains the following fields.

- **address** field holds the address of the branch instruction for which the entry corresponds to. This field is used to resolve conflicts between branch instructions whose addresses have the same low-order bits.

- **predict state** is a two bit number that stores one of the four states for the next predicted outcome of the branch. The four states are *strongly taken* (11), *taken* (10), *not taken* (01), *strongly not taken* (00). The first two states result in the branch predicted as taken. The other two states predict the branch as not taken. The transition among these states is as shown in figure 4.4.

- **predict target** this gives the target address of the branch if it would be taken.

This scheme however does not perform very well for the indirect branches. In such cases even if the outcome of the branch is predicted correctly, the target address may not be predicted correctly as it depends on the contents of memory locations. We therefore use a modified scheme for the indirect branches.

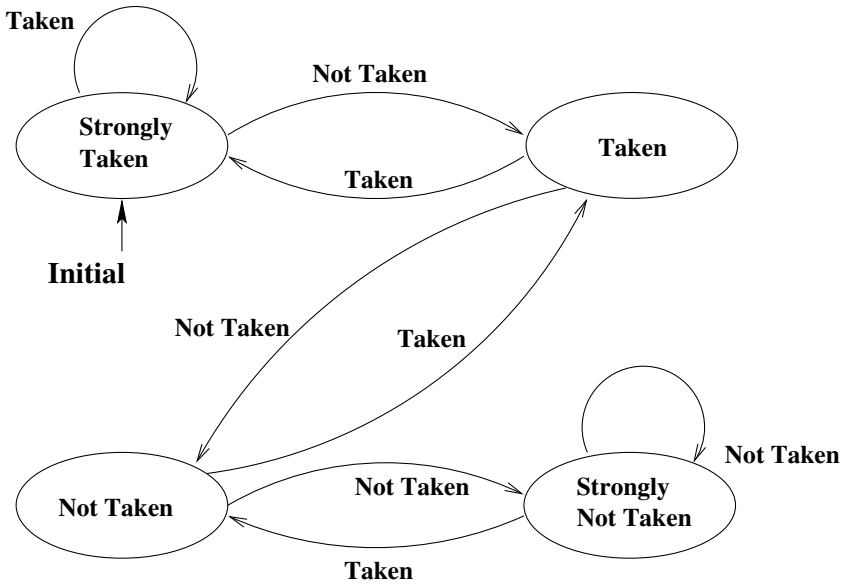


Figure 4.4: State diagram for a 2-bit branch prediction scheme

4.4.2 Indirect Branch Prediction

In the PERL programs, the most common use of indirect branches is for the function calls and return from the function. PERL programs use jump instruction to implement call to a function and return from the function. Call is performed by a direct or indirect jump in which the second operand specifies the location to store the return address. On the other hand, return from the function is performed always by an indirect jump. Since the function may be called from different locations in the program, same return instruction results in returning the control to different locations depending upon from where the function was called. This creates a problem in branch prediction. For example consider a code shown in figure 4.5. The `printf`

```
100 j _print, -8(SP)
    /* call to print storing return
       address on stack */
110 ------
120 ------
130 j _print, -8(SP)
    /* call to print storing return
       address on stack */
140 ------
150 ------

_print:
600 ------
610 ------
620 ------
700 j -8(sp), #0
    /* return by taking the return
       address from stack */
```

Figure 4.5: Example for call to the same function from different locations

function is called twice from location 100 and 130 respectively. When the first time the function returns (from location 700), no prediction is made (as there is no information for address 700). However, the BTB is updated to show the target address as 110. When the second time a return is made from the function, the target PC is predicted to be 110, whereas the actual target is 140. Thus, the target is wrongly deduced even when the prediction was right. However, in such cases the address from where call was made is related to the address where the control returns after the call is over.

This problem is solved by Kaeli and Emma [96] with a pair of stacks. In their scheme a pair of stacks S1 and S2 are used along with the BTB as shown in figure 4.6. When a jump instruction is used to call a function (100 in the example), target address (600) and the next sequential address (110) are pushed into S1 and S2 respectively. When a return instruction executes first time (700), the computed target is compared with the top of S2. If a match is found the corresponding entry

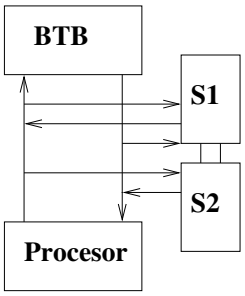


Figure 4.6: Branch prediction using a pair of stacks

in S1 is picked up and used as target address (branch address as 700 and target address as 600). For such handling, a bit is set in the BTB to indicate that the target address should be obtained from the stack rather than from the BTB.

When the call to the function is made again (at address 130 in the example), target address (600 in the example) and the next sequential address (140) are pushed into S1 and S2 respectively. When the return instruction (at location 700) is fetched, the BTB is searched for 700. At this time, an entry is found for address 700 in the BTB with the special entry bit being set. Therefore, the corresponding entry in S2 is picked up as the target address, which in this case is 140.

The processor simulation model that we use for our study implements all the features discussed here. It allows configuring various parameters of the superscalar processor and observing their effects on performance. The simulator is explained in detail in section 5.2.

4.5 Memory Subsystem

An example memory subsystem for PERL is shown in figure 4.7. We used this memory subsystem in our cache analysis done by the cache simulator. However, the cache simulator is highly configurable and can be used to experiment with several configurations.

The example memory subsystem of PERL contains an L0 cache that consists of physical registers mapped on to memory locations. L1 cache is a split cache

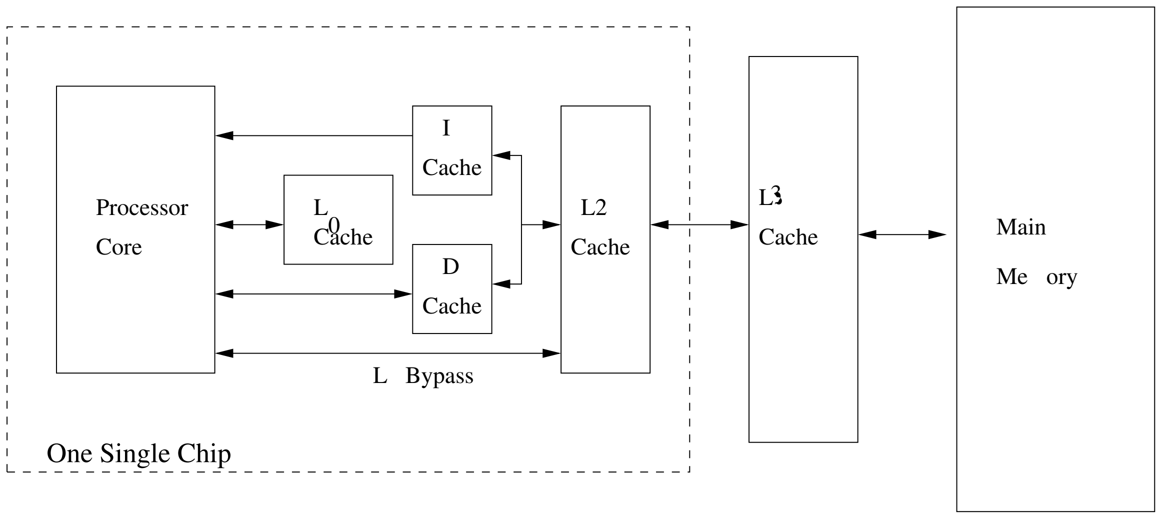


Figure 4.7: Memory hierarchy in PERL processor

between I-cache and D-cache. The bus between the I-cache and the processor is a wide bus capable of transferring multiple PERL instructions (needed by superscalar versions). The data cache is multi-ported, to enable simultaneous read of operands for multiple instructions. The L1 cache is supported by a large L2 cache. An L3 cache can also be deployed to reduce the memory latency.

In our cache simulation model, the memory subsystem consists of two primary caches, for the instruction and data respectively. The secondary cache can be configured either as unified or split. In the simulation model, there is a wide bus between instruction cache and the fetch unit capable of fetching upto 4 full instructions (aligned). Some memory locations such as stack pointer (SP), frame pointer (FP) and a small number of memory locations (e.g., temporaries) are always cached to on-chip registers (L0 cache).

The memory bandwidth requirement for PERL processor is very high. Each instruction may require up to a maximum of six memory requests: three for resolving operand een erand a 5.787 lo in Td (instructionTj 56.559 0 Td (t)Tj 9.799 0 Td 8(re52ions)Tj 401 0 Td

4.5.1 Increasing Cache Port Efficiency

Wilson et al. [58], proposed several techniques for improving cache port efficiency. Among those techniques *load-all* (LA) and *load-all-wide* (LAW) are the most suitable ones for PERL.

Load all technique increases the cache bandwidth by satisfying as many outstanding loads in parallel as possible when data is returned from the cache. The *load-all-wide* builds upon *load-all* by widening the single cache port up to the cache block size to increase cache bandwidth. All outstanding loads reside in cache access buffer. To make use of an entire cache line, each cache access buffer entry must contain a multiplexer as well as a comparator. If the comparator detects that tags are equal, then the multiplexer is used to select the correct data block from the returning cache line.

4.5.2 Multi-Ported Cache

Multiple ports are needed in the cache to meet the memory requirements of the superscalar processor. Two techniques for implementing multiple cache ports are: (i) to duplicate the cache and (ii) to interleave the cache [56].

- **Duplicate cache banks:** A straightforward way to implement multiple read ports is to provide multiple copies of the cache. For example, 4 read ports can be provided to a 16KB cache by using four caches, each of size 16KB with identical contents. This approach has a significant overhead in the amount of memory used, especially when considering an on-chip cache.

- **Interleaved banks:** A better way to provide multiple cache ports is to interleave the cache blocks amongst multiple banks, much in the same way as in an interleaved memory. A cache block is present entirely in one single cache bank.

The cache simulator that we use in our study gives the statistics about the performance of load-all-wide. It can also be configured to simulate multi-port cache

(both interleaved and duplicate banks). However, in our studies we used the interleaved bank mechanism. The details of the cache simulator are explained later in section 5.4.

Chapter 5

Simulation Methodology

There are various different ways to evaluate the benefits of design ideas for an architecture. The most accurate way is to build a prototype, which is time consuming and expensive.

Another efficient way is to build a trace-driven simulator, which uses an instruction trace generated by a trace generator. In this approach, the program trace is generated on a known machine and this trace is simulated over the machine under study. Trace-driven simulation is fast to execute, because for the simulation, only those features of the processor are modeled that affect the performance. For example, the simulator does not record the values in the register file or memory. This approach however has a disadvantage where the evolving design is not debugged because the programs are never executed. The design flaws due to cross-interacting hardware elements typically get unnoticed.

The third approach is to build an instruction set simulator. We undertook this approach in case of PERL. It offers several advantages.

1. **Accuracy**
   We wanted the results of simulated assembly instructions to be computed and the state of various hardware elements to be recorded on a cycle to cycle basis. Correctness of simulated program output was very important for us to assess the proper coordination of all the different simulated hardware components.

2. **Configurability**

   The various parameters of the architecture are made configurable in the simulator. This is important because it enables us to explore various design alternatives.

3. **Portability**

   We wanted a simulator to execute assembly language programs generated by a cross compiler, so that it would be portable to different machines.

4. **Availability**

   A superscalar simulator **superDLX** [97], was available as a free software. It is a generic superscalar processor simulator and we have re-used many parts of this simulator. In addition, we implemented the new features for simulating PERL RISC architecture. Our simulator is called **supersim**.

5.1 Evaluation Process

We chose to compare the performance of the PERL processor with that of the DLX processor mainly because of availability of its simulator. The dynamic instruction count (N) and total cycle count are the two most important performance metrics observed by us. We also give the performance of branch prediction, operand access distribution and memory subsystem to observe the details of PERL processor and its impact on the performance.

The figure 5.1, shows the step by step method adopted to evaluate the performance of PERL processor. The same steps are carried out for DLX by using *dlxcc*, **superDLX** and the appropriate configuration files.

A benchmark program is first compiled to obtain the assembly language program for PERL or DLX. The simulator takes the assembly level program, assembles, links and loads it directly into its memory. In addition, the simulator takes a processor configuration file that defines various processor parameters like order of superscalarity, size of reorder buffer, size of branch target buffer, size of instruction queue, integer and floating point instruction windows etc. A sample processor configuration file is given in appendix B.

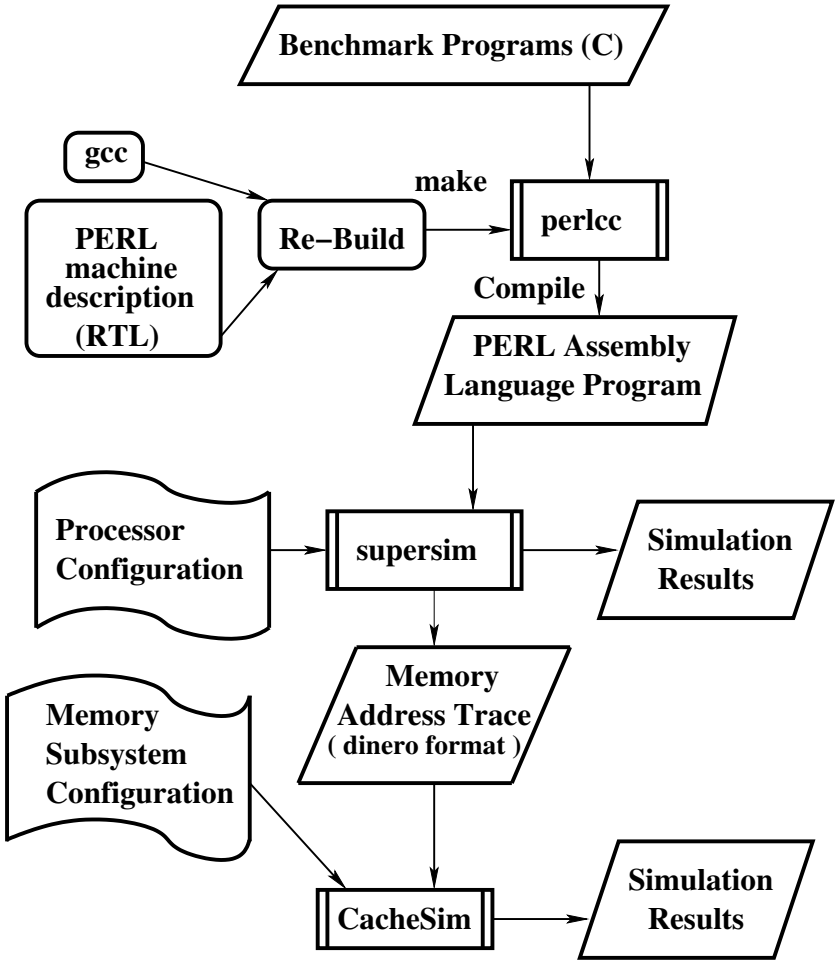


Figure 5.1: Evaluation process

During simulation, the program is executed cycle by cycle. In each cycle of the instruction, status of various processor components can be observed. Optionally we can set the simulator to generate the memory address trace in *dinero* format [22] (explained in section 5.4.1). These traces can be used off-line to evaluate the design of memory subsystem.

Several tools were used to conduct simulations, some of them were developed by us and some were freely available software. Some other freely available tools were enhanced to incorporate our requirements. The tools that were used are the

following.

1. A superscalar instruction set simulator for PERL **supersim** [98].
2. A superscalar instruction set simulator for DLX **superDLX** [97].
3. C cross-compilers for PERL (*perlcc*) [99] and DLX (*dlxcc*), both of them being port of *gcc*.
4. A trace driven cache simulator *cachesim* incorporating advanced features of multi-port caches [99]. This simulator takes the memory traces in *dinero format* [22] (explained in appendix C).

5.2 Implementation of supersim

supersim implements sophisticated and configurable superscalar instruction processing policy: multiple out-of-order issue, multiple out-of-order completion etc. To achieve this policy efficient hardware mechanisms were selected for simulation, such as, a central window to buffer decoded instructions prior to issue, and a reorder buffer, supporting operand renaming etc. Other features such as branch prediction were built around them to further boost the performance.

5.2.1 Underlying Data Structures

Supersim models PERL processor with the following two types of components.

- Memory and the functional units, which are the fundamental components of the processor.
- Other machine components such as the branch target buffer (BTB), the instruction queue, the instruction windows and reorder buffers. We collectively call them the *superscalar hardware* elements.

The simulation environment is configured using a processor configuration file (appendix B). This file describes the number of instantiations of various components, their sizes, latencies etc.

In order to execute instructions, **supersim** needs to know the details about the usage of various hardware components by the processor for each instruction. For this purpose, an instruction table is used.

5.2.2 Instruction Table

The opcode field of the instructions (figure 4.1) is used as index into the instruction table maintained by the simulator. Using this table **supersim** determines the behavior of instructions at each pipeline stage. An entry in the table reflects all the resources needed for processing of the instruction in the superscalar processor.

Each entry in the table gives the following information.

- opName: the character string for the mnemonic of the instruction.
- opUnit: the kind of the operational unit where the instruction must be processed. Only two kinds of operational units are available in the simulator, INTEGER and FLOAT.
- class: this field groups instructions into categories such as BRANCH, ALU, etc.
- funcUnit: the functional unit within an operational unit to which the instruction must be issued for execution.
- firstOperand: the type of the first operand, if it exists (NONE, INTEGER, FP_SIMPLE, FP_DOUBLE).
- secondOperand: the type of the second operand, if it exists (NONE, INTEGER, FP_SIMPLE, FP_DOUBLE).
- result: the type of the result, if it exists (NONE, INTEGER, FP_SIMPLE, FP_DOUBLE).

The following examples give the entries corresponding to instructions `add` and `addf`.

```
{"ADD", INTEGER, ALU, INT_ALU, INTEGER, INTEGER, INTEGER}
{"ADDF", FLOAT, FADD, FP_ADD, FP_DOUBLE, FP_DOUBLE, FP_DOUBLE}
```

5.2.3 Basic Processor Elements

Memory

Two kinds of memories are implemented in the PERL simulator – instruction and data.

The instruction memory is modeled as an array of the structure where each element consists of the following.

- opcode that facilitates the access to the instruction table. This field is initialized to NOP (zero).
- src1, src2, dest: The 32-bit addresses (or value when the addressing mode of the corresponding operand is immediate) of source and destination operands of the instruction.
- as1, as2, adest: Specify the addressing modes of src1, src2 and dest respectively.
- ds1, ds2, ddest: Specify the data types of src1, src2 and dest respectively.

Memory for instructions is so modeled to avoid bit manipulations and instruction decoding during simulation. As an example, the opcode, which is usually encoded in the first word of an instruction, is copied to the opcode field of the structure (similarly the addressing modes and data types).

Memory for data is modeled as an array containing memory values. The memory locations are accessed using address as index into this array. The size of the memory is a configurable parameter provided in the configuration file.

Functional Units

Functional Units are classified into two operational unit classes namely, integer and floating point.

- There are the following 5 functional units in the integer operational unit.

  - INT_ALU: for all one byte, 2 byte or 4 byte additions, subtractions and logical instructions.
  - INT_SHIFT: for all integer (signed/unsigned) shift operations.
  - INT_MUL: for integer (signed/unsigned) multiply operations.
  - INT_DIV: for integer (signed/unsigned) division operations.
  - INT_BRANCH: for all integer conditional branch and unconditional branch operations.

- There are the following 4 functional units in the floating point operational unit.
  - FP_ADD: for all floating point additions and subtractions.
  - FP_MUL: for all floating point multiply operations.
  - FP_DIV: for all floating point division operations.
  - FP_BRANCH: for all floating point conditional branch operations.

The parameters for functional units are stored in two tables in the simulator.

Nine entries in these tables (5 for the integer units in the first table and 4 for the floating point units in the second table) correspond to 9 functional units and contain the following configurable parameters.

- latency: This corresponds to the number of cycles that elapse between issue and completion of the computation. This is used to fill the ready field of an instruction in the reorder buffer.
- num units: number of available instances of that type of functional unit.
- num used: number of functional units of that type, which are currently used by the instructions in the pipeline. This parameter is evaluated dynamically, is non configurable and is used during the run time of the simulator to identify the structural hazards.

5.2.4 Superscalar Elements

This section details the mechanisms and data structures used to support the superscalar aspects of **supersim**.

Branch Target Buffer

supersim uses the 2-bit branch prediction scheme which is widely used by current day superscalar microprocessors. This mechanism helps the instruction fetch stage to predict the outcome of a conditional branch based on its past behavior and confine fetching instructions in the predicted path.

In **supersim** a table is used to model the Branch Target Buffer. The index to the table is the lower order bits of the address of the branch instruction (i.e., address modulo size of the table). The fetch stage uses contents of this table to predict a branch. Size of the BTB can be specified in the machine configuration file. The structure of the BTB and the branch prediction algorithm is implemented as described in section 4.4.1. In addition **supersim** also implements the branch prediction scheme for indirect jump (call/return), as described in section 4.4.2. A particular branch prediction scheme can be chosen for the simulation through a command line option while executing **supersim**.

Instruction Fetch Queue

The Instruction Queue is a FIFO queue where the fetch stage adds a fixed number of elements (each element corresponds to an instruction) at the top, and the decode stage takes a fixed number of elements from the bottom. Each item in the list contains the following information:

- wordPtr: a pointer to simulator's instruction memory from where this instruction is fetched.
- address: instruction address (pc).
- prediction: prediction information for branches: TAKEN, NOT TAKEN. This is the prediction made by the fetch process when it encounters a branch.

This field is used only for the branch instructions and is ignored for other instructions.

The maximum number of elements in the instruction queue (numEntries) is a user set parameter from the machine configuration file.

Instruction Windows

Instructions windows are realized as compressible stack where the decode stage adds decoded instructions in the program order. However, the issue stage can remove the instructions from anywhere in the stack (figure 5.2).

There are two instances of instruction windows in **supersim**– *iWindow* for integer instructions and *fpWindow* for floating point instructions. The sizes of the two windows can be configured by specifying them in the machine configuration file. Keeping instructions in order makes it simpler to prioritize among ready instructions, as older instructions appear at the bottom of the stack. In **supersim** the instruction windows are implemented with linked lists.

Each entry in the instruction window corresponds to an instruction that has been decoded, and contains the following fields.

- opCode: the decoded opcode of the instruction.
- class: the instruction class corresponding to the type of the functional unit.
- unit: the functional unit where the operation must take place; it is determined by the decode function (using the opcode description table).
- reorderEntry: the reorder buffer entry the instruction is assigned to.
- prediction: the prediction information (TAKEN, NOT_TAKEN) for branches.
- firstOperand: the information on the first operand and contains the following sub-fields.
  - value, which contains the operand value or the reorder buffer entry from where the value will be forwarded when computed.

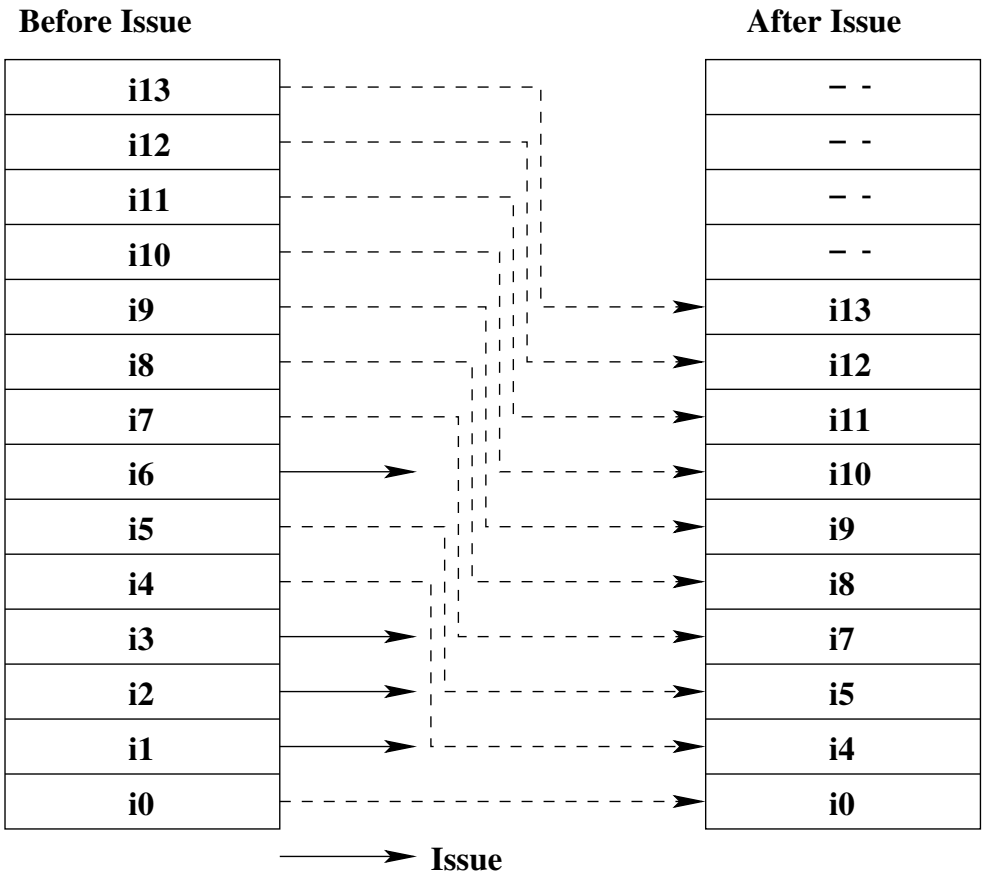


Figure 5.2: Keeping instruction in-order in the instruction window

  - type, which contains the type of the operand (integer, floating point double, floating point single).
  - valid, which indicates if the operand is ready to be used.

- secondOperand: the information on the second operand if it exists (same structure as for the firstOperand).

Reorder Buffers

Supersim maintains two reorder buffers, iReorderBuffer and fpReorderBuffer, one for the integer instructions and another for the floating point instructions. Their sizes

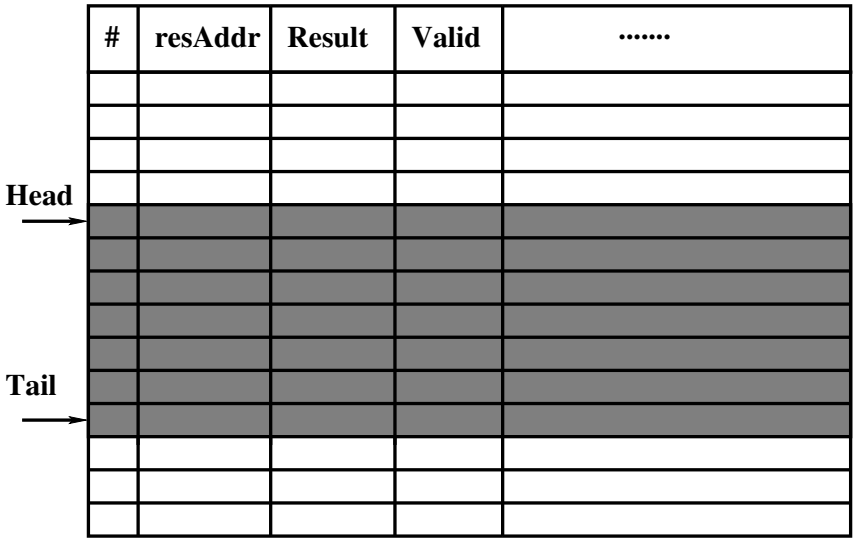


Figure 5.3: Reorder buffer entries

are configurable parameters and are specified through the machine configuration file. The buffers are implemented as circular queues with a head position (bottom of the queue) and tail (top of the queue) as shown in figure 5.3. New decoded instructions are assigned an entry at the tail of the queues, and they commit in order at the head. With this mechanism, instruction entries do not have to move towards the bottom or top of the queue, only the head and tail pointers move. Each entry in the reorder buffer consists of the following.

- OpCode: the opcode of the instruction.
- class: the class of the instruction (e.g., DIRECT_JUMP, TRAP etc.).
- resAddr: the memory address of the result.
- result: the computed result of the instruction.
- op1, op2: the value of the operand 1 and operand 2 of the instruction are stored here. In case the operand is being computed in the pipeline, it will have a pointer to the buffer entry holding the result.

- valid: the validity of the result, which indicates if the operation has been completed and result is available.
- ready: it indicates at what clock cycle the computation of the result is going to complete.
- flush: the flush bit is set if the instruction falls in a mispredicted path.
- status: cache consistency status.

A list is maintained, which links reorder buffer entries of currently executing operation. This list of pending operations (listOfExec) makes the work of the write back stage easier and faster.

5.2.5 Overall Functional Organization

Figure 5.4 gives the overall functional organization of the simulation core of **supersim**. The processes on the diagram represent groups of functions. The **supersim** is modeled on a consumer–producer relationship between processes. For example, the decode functions produce instruction information that is placed in the instruction windows and is then removed by execute functions as operations to be executed.

The **supersim** works on a per cycle basis, calling each function in a consumer-to-producer order. That is, the result commits are called first, then the write back functions etc., and this ends with the fetch process, as shown in figure 5.5 This organization prevents data from different cycles to be mixed up in the shared resources such as instruction window or the reorder buffer. But, in reality the pipeline stages are independent and communicate through shared resources (buffers).

The **supersim** simulates the processes of fetch, decode, operand fetch, execute, write back and result commit in exactly the same way as described in section 4.2.1.

The user commands are processed as shown in figure 5.6. To monitor the process of execution the **supersim** provides *step, next n* and *go* commands. To observe the status of the processor elements it has *print* command, using which one can observe the contents of instruction queue, instruction window, reorder buffers and functional units. To print the statistics of the execution one can use *stats* command.

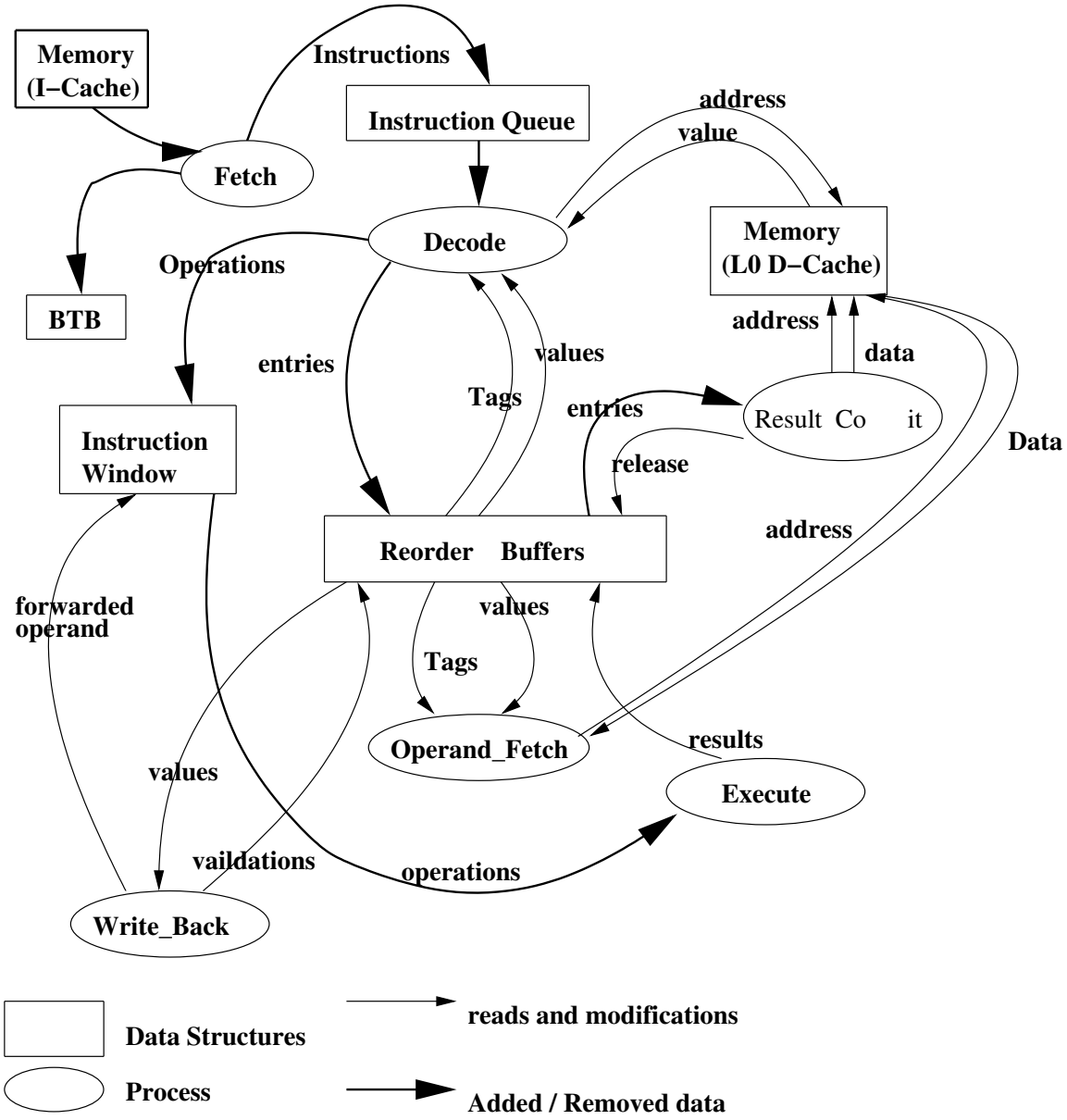


Figure 5.4: Overall functional diagram of **supersim**

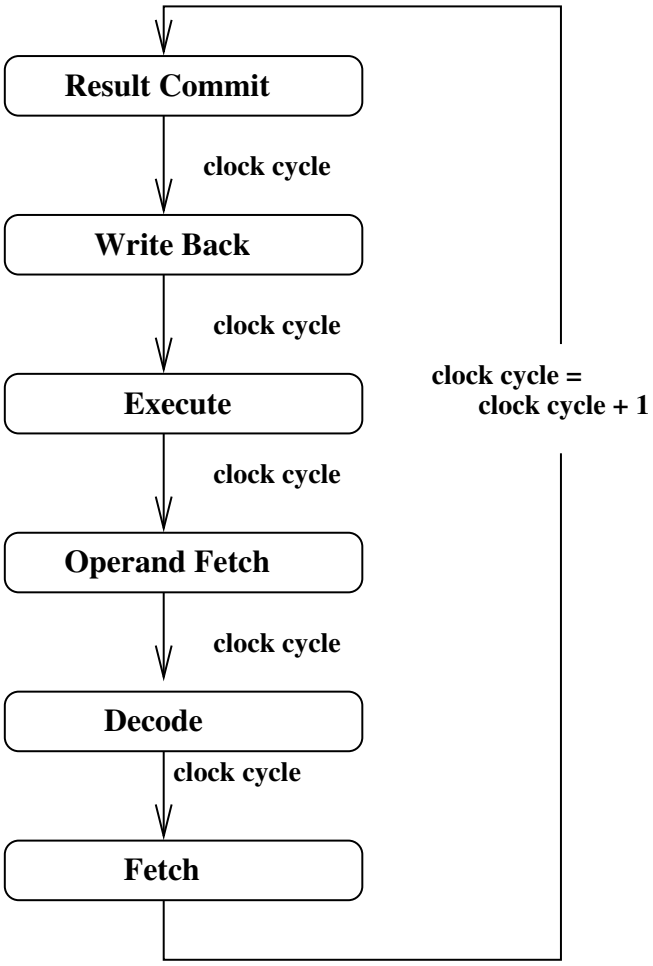


Figure 5.5: Function coordination

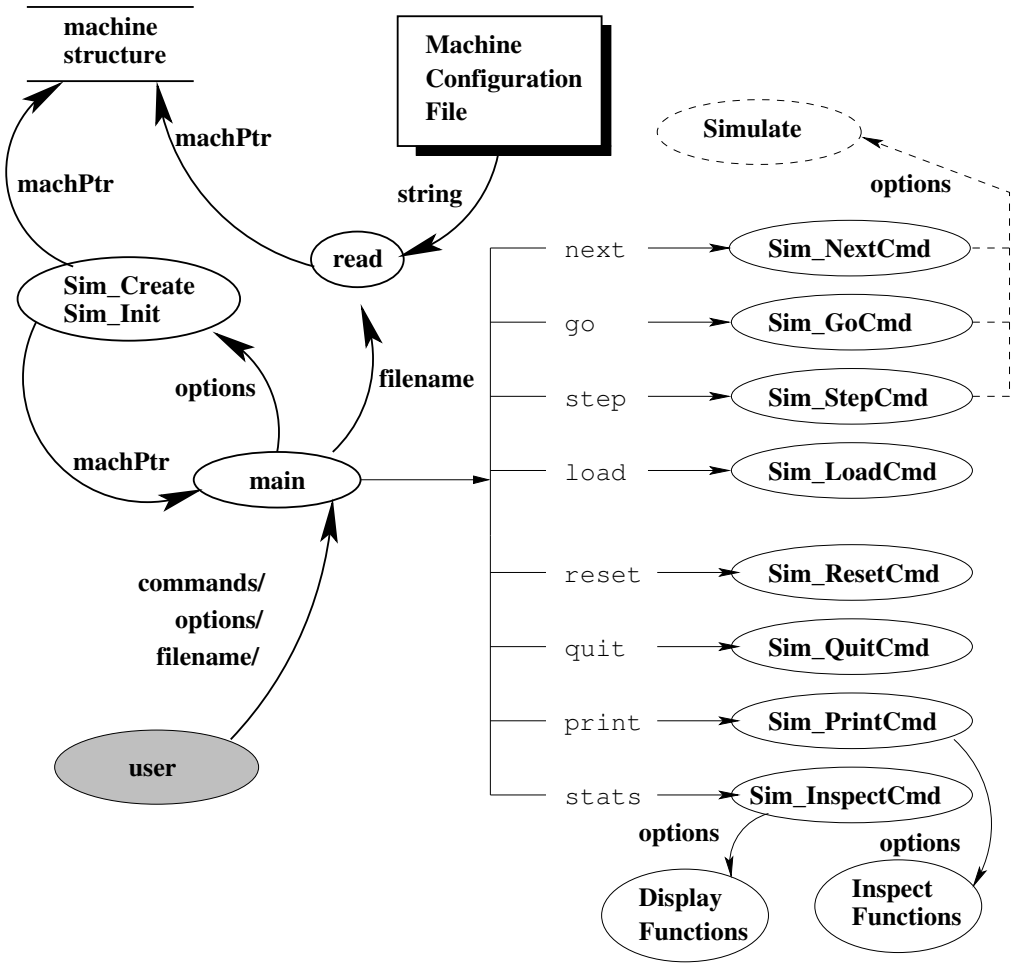


Figure 5.6: User interface functional diagram

5.3 Implementation of `perlcc`

The PERL C compiler [99] takes *C* language program as input and produces assembly code of PERL as output. It is built upon *GNU C* [100], a fast and highly portable compiler available in source form. GCC gets most of the information about the target machine from a machine description which provides templates for each of the machine's instruction.

5.3.1 Machine Description

A machine description has two parts: a file of instruction patterns ('.md' file) and a C header file of macro definitions. The '.md' file for a target machine contains a pattern for each instruction supported by the target machine. Information about the target machine architecture such as registers, addressing modes, stack organization etc. is supplied in a C header file. In our compiler, we have not implemented any machine dependent optimizations.

Instruction Patterns

Each instruction pattern contains an incomplete parameterized RTL expression, with pieces to be filled in later, operand constraints that restrict how pieces can be filled in, and an output pattern or C code to generate the assembler output, all wrapped up in a define_insn expression.

5.3.2 Machine Description for PERL

Architecture Specification

Storage layout

The processor is defined as *big endian*, most significant byte in a word has the lowest address. In a multi-word the most significant word has the lowest address. The least addressable storage unit is a byte, which has eight bits. Word size and the addresses are 32 bits. Function entry points and instruction addresses such as branch target

are aligned on sixteen byte blocks (instruction boundaries), to fetch an instruction in a single read request. All other objects are aligned on word boundaries.

Temporary locations usage

The number of registers of the processor, and their usage is supplied to GCC through C Macro definitions. Compiler needs at least two registers to be specified, stack pointer (SP) and frame pointer (FP). In order to access indirect operands, we use temporary locations in PERL. As GCC has no concept of temporary locations, these are faked as registers in the *.md* file. Assembly code declares temporaries for global references. Size of each temporary location is four bytes. Currently we are using sixteen such temporary variables (including SP and FP).

Stack layout

The PERL has no hardware stack. The stack has to be implemented in software itself. GCC allocates space for local variables of a function in its frame. The frames for the functions are allocated on the stack.

In our implementation, we use the frame in a certain format. First three words (12 bytes) on the frame have the following fixed usage.

- The first word (4 bytes) is used to store the old frame pointer at the function entry point.
- The next word is used to store the return address from the function.
- The last word is used to pass the function results to the caller function. It contains the address in memory where the return value is present.

In PERL, there is no function call instruction. Stack adjustment is done by the explicitly generated assembly code, both at the function entry and exit points.

Instruction Patterns

All the available assembly instructions in PERL are specified to the compiler using instruction pattern.

For the purpose of specifying instruction patterns, the assembly instructions can be broadly classified into two types: arithmetic and flow control instructions.

Arithmetic instructions

For each available assembly instruction, a named define_insn pattern is specified.

For example the addb4 instruction is specified by using the following instruction pattern.

```
(define_insn "addsi3"
  [(set (match_operand:SI 0 "general_or_addr_operand" "")
        (plus:SI (match_operand:SI 1 "general_or_addr_operand" "")
                 (match_operand:SI 2 "general_or_addr_operand" "")))]
  ""
  "*
  {
         return \"addb4 %0,%1,%2\";
  }")
```

The function general_or_addr_operand checks the operand addressing mode. In case of a valid addressing mode, the functions matches the instruction pattern.

Control instructions

GCC assumes that the machine has a condition code. A comparison instruction sets the condition code, recording the results of both signed and unsigned comparison of the given operands. A separate branch instruction tests the condition code and branches or not according its value.

PERL has compare-and-branch instructions and has no condition code. As there is no assembly instruction in PERL corresponding to a comparison instruction generated by the compiler, a define_expand expression is specified to record the operands in two static variables.

For example a define_expand expression for integer comparison instruction is specified as follows.

```
(define_expand "cmpsi"
  [(set (cc0) (compare
                (match_operand:SI 0 "general_or_addr_operand" "")
                (match_operand:SI 1 "general_or_addr_operand" "")))]
```

```
  ""
  "
{
  compare_op0 = operands[0];
  compare_op1 = operands[1];
   DONE;
}")
```

The **DONE** macro in C preparation statements specifies that no RTL code will be generated for this instruction. When outputting the branch-on-condition-code instruction that follows, the compiler actually outputs a compare-and-branch instruction that uses the remembered operands.

For example, a branch-on-equal instruction is specified as follows.

```
(define_insn "bge"
  [(set (pc)
        (if_then_else (eq (cc0)
                          (const_int 0))
                      (label_ref (match_operand 0 "" ""))
                      (pc)))]
  ""
  "*
{
  operands[1] = compare_op0;
  operands[2] = compare_op1;
  return \"jeqb4 %l0,%1,%2\";
}
```

For call instructions an unconditional jump instruction, j function, -8(sp), is generated. The jump instruction stores the return address on the stack, which is later used to return from the function.

5.4 Implementation of Cache Simulator

The cache simulator is a modified version of an existing cache simulator [101]. It was modified to simulate 'Load All Wide' technique and multi-port caches. It can be used either as a stand alone trace-driven simulator or along with an Instruction set simulator to simulate entire memory-hierarchy. In our simulations, we use the simulator in stand-alone trace-driven mode. We describe the details of the simulator in this section.

5.4.1 Simulator Input

The input to the simulator is a memory trace file and a cache configuration file. The format of these files is given in appendix C.

The trace file is generated by the processor instruction set simulator while executing a program. For each reference, a trace contains the following information in addition to the address of the memory reference.

- Type of reference. Specifies whether the reference is fetch, read or write.
- Clock cycle. Specifies the processor clock cycle with reference to the start of execution in which the reference is made.

The configuration file supplies different parameters of cache hierarchy. It specifies number of levels of cache hierarchy and the following for each level of the cache.

- Type of cache: Unified or split cache. In case of split cache, the cache parameters are specified separately for both Instruction and data cache. In case of Unified cache, these parameters are specified only once.
- Number of cache lines.
- Line size.
- Associativity.
- Number of interleaved cache ports.

- Number of duplicate cache ports.
- Write policy, only in case of Unified or data cache. This can be either write back or write through.
- Number of clock cycles required to satisfy the request, in case of a cache miss at the current level but a hit at the next level.

All parameters of the cache can be configured except the cache block replacement policy, which is fixed as *least recently used* in the simulator.

5.4.2 Simulator Output

The simulator gives the following performance metrics for each level of cache.

- Miss-ratio for each category of misses.
- Total number of write backs required in case of a write back cache.

In addition the simulator also gives the number of references that are served by *load-all-wide* optimization and the worst case clock cycles required for execution of the program. Worst case clock cycles are calculated by adding penalties due to cache bank clashes and misses. Simulator serves references in a particular clock cycle only after serving all references of the previous clock cycle.

Chapter 6

Results

Extensive simulations were carried out to study the performance of the PERL processor. The exact steps involved in the simulations have already been explained in chapter 5. We compared the performance of the PERL processor with that of the DLX processor. The summary of the results are presented here.

Simulations were performed on a collection of seven benchmark programs. First, we discuss the general performance trends exhibited by all these programs such as program size and dynamic instruction count. Following this, we discuss the results of simulations on each of the benchmark programs.

6.1 Benchmark Programs

We have performed simulations on programs taken from several users in our lab, and from SPEC95 [102] and NASA Numerical Aerodynamic Simulation (NAS) [103] test suites. The following programs were simulated and the simulation results are presented here.

1. **Permute.** This is a highly recursive program, which given a positive integer **n**, computes all **n**! permutations of numbers from 1 to **n**. For simulations we have taken **n** as 5. This is a very tiny program and is CPU-intensive.

2. **relax**, **across** and **mult**. These three programs are taken from NASA NAS test suite for parallelizing compilers. All of them contain nested do loops and

operate on vectors. The original codes of these programs are in FORTRAN and the UNIX `f2c` utility is used to convert them to C code.

3. **ttn**. This is a timetable scheduler program. Given a list of courses, preferences of timing for allotting slots to the course and a given set of class rooms (it takes this information from two files), the program uses a heuristic approach to get the timetable schedule. This program uses some floating point instructions (about 10.5% and 5.4% of dynamic instruction count in DLX and PERL respectively). This program is CPU intensive but also performs I/O.

4. **compress.** This is a standard benchmark program from SPEC95. It performs the standard compress and decompress (similar to UNIX utilities) over a set of randomly generated files. It also compares the decompressed file with the original file to verify the results of compression and decompression. The files are generated directly in memory and hence *compress* benchmark performs no disk access. This benchmark is a CPU-bound integer program, but also uses a small number of floating point instructions (about 3% and 1.7% of dynamic instruction count in DLX and PERL respectively).

5. **go.** This is a standard benchmark program from SPEC95. It is an example of the use of artificial intelligence in game playing. This program plays the game of go against itself. The benchmark is stripped down version of a successful go-playing computer program. This benchmark is a CPU-bound integer benchmark.

Thus, these programs represent a variety of execution behaviors ranging from I/O bound jobs to compute bound jobs, small to large programs, and memory intensive to non-memory intensive programs.

6.2 Machine Models

We carried out simulations for three variations of PERL and DLX each. The variations are non-superscalar (*md1*), superscalar of order 2 (*md2*) and superscalar of

order 4 (*md4*). The number of arithmetic units for corresponding PERL and DLX models are kept the same and is shown in table 6.1.

As PERL is a memory-to-memory processor, we provided it the capability of 2 memory accesses (to D cache) per clock for *md1* and 3 memory accesses (to D cache) per cycle for *md2* and *md4*. The question that immediately arises is whether the DLX would perform better given the same memory bandwidth or not. Therefore, DLX is also provided with the same configuration of memory (i.e. 2-port L1 cache in *md1* and 3-port L1 cache in *md2* and *md4*). Therefore, even though the *md1* configuration of DLX machine is a non superscalar model, it has the capability to issue simultaneous memory accesses. Similarly, the *md2* and *md4* configuration can issue three simultaneous memory accesses.

PERL has a small number of registers that are mapped onto memory locations (L0 cache). L0 cache is expected to cache all temporary variables including the SP, FP and the base address holders specified using a shorthand representation.

In addition to the above variations, branch prediction schemes are used in both PERL and DLX simulators. DLX has a branch prediction mechanism using a 2-bit branch history (*bp*). PERL has four branch prediction schemes, namely, branch prediction using a 2-bit branch history (*bp*), indirect branch prediction using a 2-bit branch history (*bi*), indirect branch using a single stack (*bs*) and indirect branch using a pair of stacks (*bS*). The branch prediction schemes *bi*, *bs* and *bS* perform direct branch prediction using *bp* scheme in addition to predicting indirect branches using the respective schemes (using the same BTB). Therefore in all we have six machine configurations of DLX and fifteen machine configurations of PERL.

The six machine configurations of DLX are *md1*, *md2* and *md4* each with no branch prediction (*nbp*) and branch prediction (*bp*).

The fifteen machine configurations of PERL are *md1*, *md2* and *md4* each with the following branch prediction schemes.

1. No branch prediction (*nbp*).
2. Branch prediction using (*bp*).
3. Branch prediction using (*bi*).

param	DLX			PERL		
	md1	md2	md4	md1	md2	md4
Maximum number of Instruction_Processed_Per_Cycle						
fetch	1	2	4	1	2	4
decode	1	2	4	1	2	4
issue	1	2	4	1	2	4
write back	1	2	4	1	2	4
commit	1	2	4	1	2	4
Buffer size						
IQ_size	32	32	32	32	32	32
int_IW_size	32	32	32	32	32	32
flt_IW_size	32	32	32	32	32	32
int_rob_size	32	32	32	32	32	32
flt_rob_size	32	32	32	32	32	32
BTB_size	111	111	111	111	111	111
No. of functional units (their latenices)						
int_alu	1(1)	2(1)	4(1)	1(1)	2(1)	4(1)
int_shift	1(1)	1(1)	2(1)	1(1)	1(1)	2(1)
int_comp	1(1)	1(1)	1(1)	1(1)	1(1)	1(1)
int_addr	2(1)	3(1)	3(1)	–	–	–
int_branch	1(1)	1(1)	1(1)	1(1)	1(1)	1(1)
flt_add	1(2)	2 (2)	2(2)	2(2)	2(2)	2(2)
flt_mult	1(5)	1(5)	1(5)	1(5)	1(5)	1(5)
flt_DIV	1(8)	1(8)	1(8)	1(8)	1(8)	1(8)
flt_convt	1(2)	1(2)	1(2)	–	–	–
flt_comp	1(1)	1(1)	1(1)	1(1)	1(1)	1(1)
flt_addr	2(1)	3(1)	3(1)	–	–	–
flt_branch	1(1)	1(1)	1(1)	1(1)	1(1)	1(1)

Table 6.1: Parameter values used in different variations of DLX and PERL

4. Branch prediction using (*bs*).

5. Branch prediction using (*bS*).

Each benchmark program was simulated on all configurations of PERL and DLX.

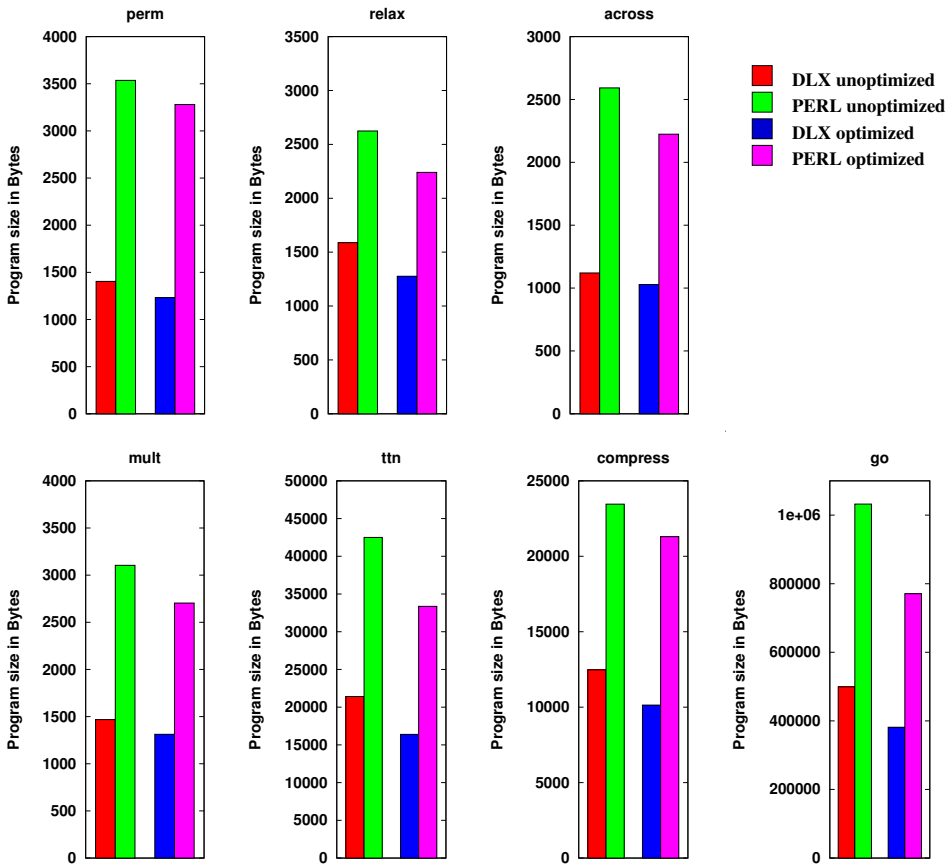
We used *dlxcc* and *perlcc* compilers to perform machine independent optimizations. The DLX compiler also performs many machine dependent optimizations. The *perlcc* compiler, however, does not perform any machine specific optimizations. In case of some programs (*ttn*, *compress* and *go*), *dlxcc* was not able to perform any machine dependent optimizations. It should be noted that in general the quality of the code generated by *dlxcc* is better than that of the code generated by *perlcc*.

6.3 General Observation

We first present some basic characteristics of PERL architecture and compare it with that of DLX. We present two important characteristics namely the program size and dynamic instruction count. These characteristics have no dependence with the simulator, however they strongly depend on the compiler and the processor instruction set.

6.3.1 Program Size

To start with, we expected the number of instructions for a program to be smaller in PERL than in DLX as PERL does not require Load and Store instructions. However, we were not sure whether this would result in a smaller program size for PERL than that for the DLX. A program is typically composed of data, code and some blank space introduced by the compilers to align addresses of variables. Contribution in program size due to data is the property of program and remains the same in both DLX and PERL. The blank space due to data alignment is also roughly the same for both processors. Therefore, it is ideal to consider only the code size of each program. The plots and table in figure 6.1 show the program size for all benchmark programs. The figures are presented for both optimized and unoptimized code.



Program	Unoptimized			Optimized		
	DLX	PERL	factor	DLX	PERL	factor
perm	1404	3536	2.51	1232	3280	2.66
relax	1588	2624	1.65	1276	2240	1.75
across	1120	2592	2.31	1028	2224	2.16
mult	1468	3104	2.11	1312	2704	2.06
ttn	21416	42496	1.98	16396	33360	2.03
compress	12476	23456	1.88	10136	21300	2.10
go	499064	1032160	2.07	381484	770832	2.02

Figure 6.1: Static code size of programs in DLX and PERL

It is seen from the plots in figure 6.1 that the static sizes of PERL programs are approximately twice the corresponding sizes of DLX programs. This was expected as size of each PERL instruction is 4 times that of the size of a DLX instruction. The program size of PERL varies from 1.65 to 2.52 times the size of DLX programs.

6.3.2 Dynamic Instruction Count

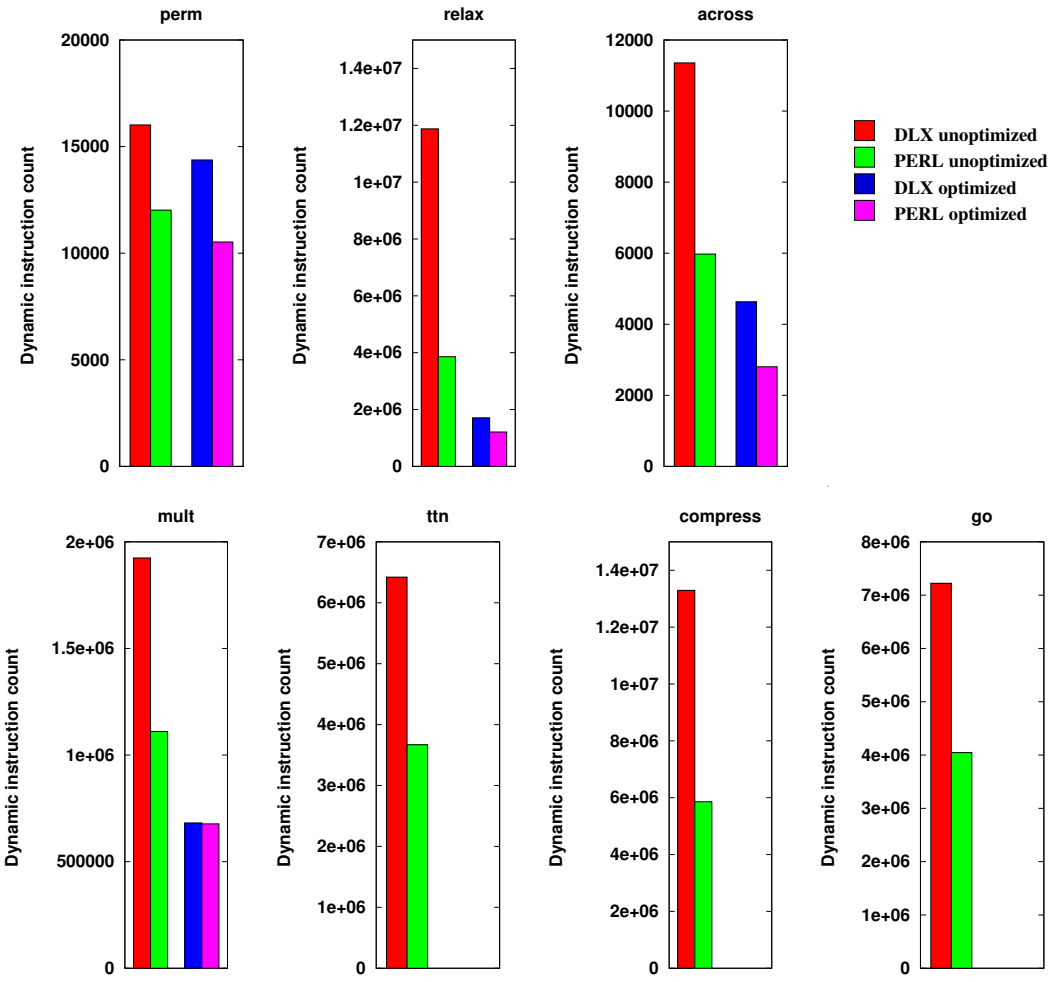
We expected that number of instructions that are required to execute a program (N) in PERL would be at least 30% less than those in DLX, as explained in chapter 3. The dynamic instruction counts for all benchmark programs are shown in the table and plots in figure 6.2.

PERL consistently executes fewer instructions compared to the DLX across all benchmarks both in unoptimized and optimized codes. The actual reduction in the instruction varies from program to program as shown in figure 6.2. The optimized code of *mult* has only about 0.6% more instructions in DLX than those in PERL – a notable exception where PERL compiler has generated a very bad code. The unoptimized code of *relax* results in the execution of three times more instructions in DLX than those in PERL. On the other hand, the difference in the number of instructions executed for the optimized and unoptimized code for *perm* benchmark is very small. The optimized code of DLX requires significantly fewer instructions to execute than those required for its unoptimized code.

This is an important result for us in making the case for PERL. This was one of the very important factors, which motivated us to study PERL, the register–less processor.

6.4 Performance Results and Analysis

In this section, we present the performance results obtained from the simulation carried out for each benchmark program. In particular, we discuss the execution time of program, extracted ILP, performance boost from branch prediction etc. We also discuss the fetch stalls due to filled up instruction queue.



Program	Unoptimized			Optimized		
	DLX	PERL	factor	DLX	PERL	factor
perm	16016	12020	1.33	14376	10524	1.37
relax	11875491	3858898	3.07	1708895	1210018	1.41
across	11355	5975	1.90	4634	2804	1.65
mult	1924406	1110566	1.73	681760	677413	1.006
ttn	6420407	3669564	1.75	–	–	–
compress	13291927	5856103	2.26	–	–	–
go	7222377	4046282	1.78	–	–	–

Figure 6.2: Dynamic instruction counts in DLX and PERL

Since each program exhibits unique behavior with respect to these metrics, we discuss these issues by taking one program at a time.

While executing the programs the simulator does not account for any memory stalls, instead it assumes a perfect cache (100% hit). It also does not model any I/O (the code contains a trap instruction for every system/library call). The trap instructions to system calls/library calls are implemented in the simulator by executing code on the host machine.

The results of the execution driven simulation are presented in this chapter while the results of off-line memory simulations are presented in chapter 7.

6.4.1 perm benchmark

The *perm* benchmark program is a permutation program which uses recursion extensively and performs no input output. Given **n** it computes all **n**! permutation of natural numbers from 1 to **n**. For simulation we take **n** as 5.

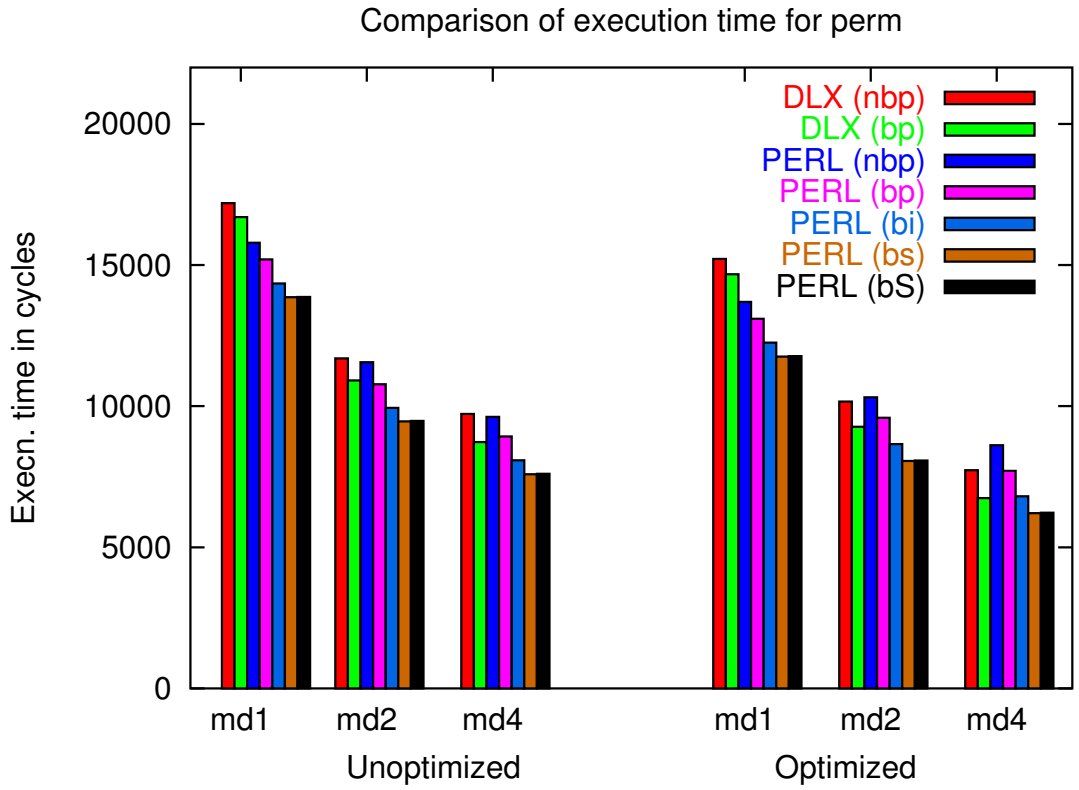
The plots and table in figure 6.3 give the execution time of *perm* on all the simulation models of DLX and PERL. The results are presented for both unoptimized and optimized code.

The plots in the figure 6.3 give the general trend that we also observed across all other programs. The best performance is always obtained on *md4* with branch prediction (*bp* in DLX and *bs* in PERL). However, the scale of performance gain from *md1* to *md4* and the effect of branch prediction vary according to the execution behavior of the program.

The execution time of *perm* in PERL is consistently better than that in DLX across all models and is true for both unoptimized and optimized codes.

The performance of various branch prediction schemes used in DLX and PERL is shown in table 6.2. It may be noted that the number of branches (417: direct and 447: indirect) executed in DLX and PERL is identical (as expected and assumed in the analytical model described in section 3.6).

The *perm* program uses recursive calls extensively and hence the effect of indirect branch prediction is reflected in the execution time. The execution time is the least with branch prediction schemes *bs* and *bS*. The performance of the branch prediction



Config	DLX		PERL				
	nbp	bp	nbp	bp	bi	bs	bS
Unoptimized code							
md1	17190	16700	15790	15199	14351	13859	13873
md2	11695	10912	11559	10777	9938	9465	9480
md4	9725	**8725**	9623	8931	8083	**7590**	7605
Optimized code							
md1	15215	14673	13696	13099	12251	11759	11773
md2	10165	9274	10315	9591	8659	8059	8076
md4	7728	**6740**	8620	7709	6810	**6210**	6227

Figure 6.3: Execution time in cycles for *perm*

Program	Metric	DLX-*md4*	PERL-*md4*			
		bp	bp	bi	bs	bS
		D_BR	D_BR	IND_BR	IND_BR	IND_BR
perm	Total #	417	417	447	447	447
	Correct	197	197	208	447	442
	Succ.rate	47.24%	47.24%	46.53%	100%	98.88%

Table 6.2: Success rate of different branch prediction schemes in *perm*

scheme *bi* is not the same as that of *bs* or *bS* because the scheme may lead to wrong predictions as explained in section 4.4.2

The performance of *bp* branch prediction scheme is same in both DLX and PERL with a success rate of 47%. However, *perm* has a significant number of indirect branches due to recursive calls, which are predicted with an accuracy close to 100% in both *bs* and *bS*. The small difference in the number of correct predictions in *bs* and *bS* are due to returns from first time calls which are not predicted correctly in *bS*. The success rate of *bi* scheme of branch prediction is poor (46.5%) for the reason explained in section 4.4.2. The number of collisions in BTB is zero in both DLX and PERL as the program is small and contains very few branches. The *bi*, *bs* and *bS* schemes predict direct branches using the same scheme as in *bp* and show the same accuracy for *perm* benchmark.

The processor cannot fetch any instructions if it has no space in the instruction queue or when it cannot determine the address of the next instruction (Fetch Stall cycles). The number of fetch stall cycles reduces when the branch prediction is used and the reduction depends on the accuracy of branch prediction scheme. As *perm* executes many indirect branches, the percentage of fetch stall cycles is more in DLX than in PERL. The table 6.3 shows the number of fetch stall cycles in both DLX and PERL for the model *md4* with all branch prediction schemes used.

The number of fetch stall cycles due to filled up instruction queue is observed to be smaller in DLX than that in PERL.

The processor has to introduce decode stall if the instruction queue is empty or if there is data dependency between the instruction being decoded and another

DLX-*md4*		PERL-*md4*				
nbp	bp	nbp	bp	bi	bs	bS
Fetch stalls in Opt. code of *perm* on *md4*						
3776	2273	5586	3999	498	984	1029
48.86%	33.72%	64.8%	51.87%	7.31%	15.85%	16.52%
Decode stalls in Opt. code *perm* on *md4*						
3211	1928	5228	4021	2909	2608	2624
41.55%	28.61%	60.65%	52.16%	42.72%	42.0%	42.14%

Table 6.3: Fetch and decode stall cycles in *perm* (percentage of total cycles)

instruction already in execution (Decode Stall cycles). We show the number of decode stall cycles in table 6.3 for both DLX and PERL. The figures are shown only for model *md4* with all branch prediction schemes used. Due to the presence of large number of indirect branches in *perm*, DLX has more decode stall cycles than PERL.

An important observation here is that *bi* scheme gives a better reduction in fetch stall cycles. However, the same reduction is not shown in decode stall cycles. This is due to poor success rate of indirect branch prediction that has forced PERL to fetch instructions in wrong path.

The number of fetch and decode stall cycles increases from model *md1* to *md4* in both DLX and PERL across all benchmark programs.

The average number of Instructions Processed per Cycle (IPC) is shown in table 6.4 at each of the fetch (FPC), decode (DPC), issue (EPC) and commit (CPC) stages. The figures are given for all the models with branch prediction (*bp* for DLX and *bs* for PERL), because the machines perform best with these branch prediction schemes.

The relation $\mathbf{FPC} \geq \mathbf{DPC} \geq \mathbf{EPC} \geq \mathbf{CPC}$ is observed. This is due to speculative instruction processing. Speculatively fetched instructions may not be decoded if speculation is determined to be wrong just prior to decoding. Similarly, all instructions, which are decoded may not execute and all instructions executed may not commit. Hence, at every successive stage there may be some instructions which

Model	FPC	DPC	EPC	CPC
Unoptimized Code				
DLX-md1-bp	0.99	0.97	0.93	0.93
PERL-md1-bs	0.98	0.88	0.87	0.87
DLX-md2-bp	1.63	1.59	1.59	1.47
PERL-md2-bs	1.81	1.34	1.27	1.27
DLX-md4-bp	2.35	2.14	2.09	1.84
PERL-md4-bs	2.51	1.71	1.59	1.58
Optimized Code				
DLX-md1-bp	0.99	0.98	0.95	0.95
PERL-md1-bs	0.98	0.91	0.90	0.89
DLX-md2-bp	1.62	1.59	1.56	1.55
PERL-md2-bs	1.87	1.39	1.31	1.31
DLX-md4-bp	3.14	2.34	2.23	2.13
PERL-md4-bs	2.83	1.86	1.73	1.69

Table 6.4: Instructions fetch, decode, issue and commit per cycle (for *perm*)

are discarded due to wrong speculation. In case of *perm* the success rate of direct branch prediction is about 47% and its effect is observed on the IPC.

Data dependency between instructions also reduces IPC. However, in case of data dependency only the instruction processing is stalled, and the instructions are not flushed.

An interesting observation is that the performance of unoptimized code for PERL is better than the optimized code for the DLX. This is due to the facts that the PERL requires fewer instructions and uses indirect branch prediction.

6.4.2 relax benchmark

The relax benchmark is a small program and performs no input output. This program uses a single matrix, initializes it to zero, and performs complicated transformations using a nested `for` loops of depth three. The matrix is declared as a local variable. The key feature in this program is the order in which the elements of the matrix are accessed.

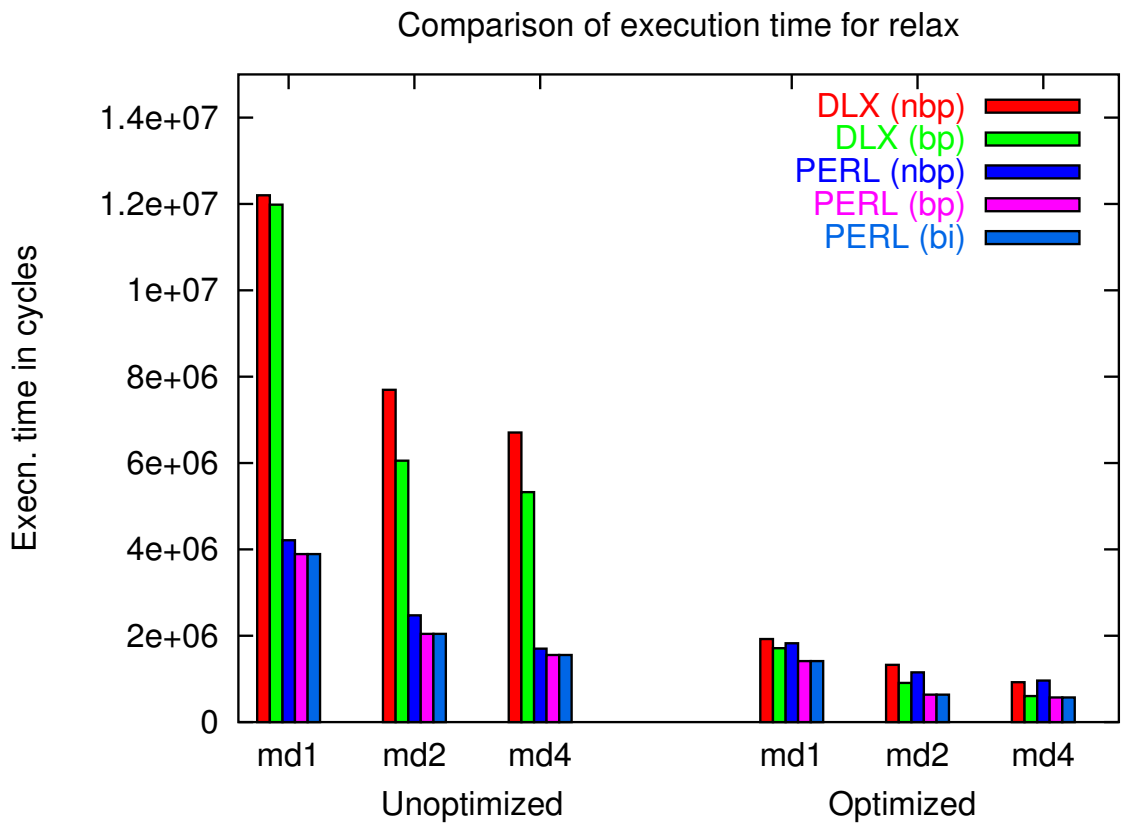
The bar charts and the table in figure 6.4 give the execution time of *relax* on all the simulation models of DLX and PERL. The results are presented for both unoptimized and optimized code.

The least execution time is observed in *md4* model with branch prediction (*bp* in DLX and *bs* in PERL). However, in *relax* there is exactly one indirect branch due to a return from library function `printf`, which is executed 10 times. Hence, it can be observed that there is no significant impact of indirect branch prediction schemes on the execution time.

An interesting feature in figure 6.4 is that for the unoptimized code the speedup obtained in both DLX and PERL for models *md1* to *md4* is almost linear. This gives an indication that there is good amount of ILP available in the program. The same kind of scaling is observed for optimized code as well. The speedup obtained by PERL in unoptimized code is more pronounced primarily because of the instruction count, where the number of instructions executed by DLX is about 3 times than that by PERL. In optimized code this is considerably reduced and the number of instructions executed by DLX is about 1.4 times than that by PERL.

The performance of various branch prediction schemes used in DLX and PERL is shown in table 6.5. It can be seen that the number of branches executed in DLX is identical to that in PERL (as expected and assumed in the analytical model described in section 3.6). The performance of *bp* is same in both DLX and PERL with a success rate of 99%. The only indirect branch present in the program is executed 10 times, the *bs* branch prediction scheme predicts it with a 100% accuracy. Whereas *bS* and *bi* schemes predict it with 90% accuracy, the only misprediction occurs when the branch is encountered for the first time.

The number of fetch and decode stall cycles in both DLX and PERL are shown


Comparison of execution time for relax
Execn. time in cycles
1.4e+07
1.2e+07
1e+07
8e+06
6e+06
4e+06
2e+06
0
DLX (nbp)
DLX (bp)
PERL (nbp)
PERL (bp)
PERL (bi)
md1
md2
md4
md1
md2
md4
Unoptimized
Optimized

Program	Metric	DLX-*md4*	PERL-*md4*			
		bp	bp	bi	bs	bS
		D_BR	D_BR	IND_BR	IND_BR	IND_BR
relax	Total #	108021	108021	10	10	10
	Correct	106925	106925	9	10	9
	Succ.rate	98.99%	98.99%	90%	100%	90%

Table 6.5: Success rate of different branch prediction schemes in *relax*

DLX-*md4*		PERL-*md4*				
nbp	bp	nbp	bp	bi	bs	bS
Fetch stalls in Opt. code of *relax* on *md4*						
422289	1127	653713	156416	156398	156396	156398
45.58%	0.19%	67.77%	27.32%	27.31%	27.31%	27.31%
Decode stalls in Opt. code of *relax* on *md4*						
422285	3312	558643	260605	260587	260585	260587
45.58%	0.55%	57.91%	45.51%	45.51%	45.51%	45.51%

Table 6.6: Fetch and decode stall cycles in *relax* (percentage of total cycles)

in the table 6.6. As expected, the number of fetch and decode stall cycles reduce when branch prediction schemes are used. The reduction in fetch and decode stalls in DLX is more pronounced than in PERL. However, this reduction is not due to branch prediction alone. While executing *relax*, 98% of the fetch stalls in PERL are due to filled up instruction queue, whereas there are no such fetch stalls observed in DLX. The number of decode stall cycles are significantly more than the number of fetch stall cycles in both DLX and PERL. There is no significant impact of indirect branch prediction scheme as indirect branches are executed only ten times.

The table 6.7 shows the average number of instructions processed per cycle. Both PERL and DLX achieve good numbers here. An interesting observation is that while executing unoptimized code DLX shows better IPC in *md1* and *md2* while PERL has better IPC in *md4*. While executing the optimized code, DLX has better IPC in *md1* and *md4*, where as PERL has better IPC in *md2*. This indicates that the

relax benchmark has good ILP in it, which has been exploited by both DLX and PERL.

An interesting observation is that the performance of unoptimized code for PERL is better than the optimized code for DLX. PERL performs consistently better than DLX while executing both unoptimized and optimized code of *relax*. This is because, PERL requires fewer instructions than DLX, has got high rate of success in branch prediction, and has an ILP of 2.1.

Model	FPC	DPC	EPC	CPC
Unoptimized Code				
DLX-md1-bp	1.0	0.99	0.99	0.99
PERL-md1-bs	0.99	0.99	0.99	0.99
DLX-md2-bp	1.98	1.97	1.96	1.96
PERL-md2-bs	1.89	1.89	1.89	1.89
DLX-md4-bp	2.26	2.24	2.24	2.23
PERL-md4-bs	2.51	2.49	2.29	2.48
Optimized Code				
DLX-md1-bp	1.0	1.0	1.0	1.0
PERL-md1-bs	0.88	0.86	0.86	0.86
DLX-md2-bp	1.89	1.89	1.89	1.89
PERL-md2-bs	1.92	1.91	1.90	1.90
DLX-md4-bp	2.88	2.83	2.83	2.83
PERL-md4-bs	2.18	2.12	2.11	2.11

Table 6.7: Instructions fetch, decode, issue and commit per cycle (for *relax*)

6.4.3 across benchmark

The across benchmark is also a very small program with no input output. This program initializes five vectors to zero and performs some simple computation using

Program	Metric	DLX-*md4*	PERL-*md4*			
		bp	bp	bi	bs	bS
		D_BR	D_BR	IND_BR	IND_BR	IND_BR
across	Total #	201	201	0	0	0
	Correct	197	197	—	—	—
	Succ.rate	98.01%	98.01%	—	—	—

Table 6.8: Success rate of different branch prediction schemes in *across*

them in a single for loop. The feature of this program is the data dependency among the instructions in the loop (intra loop data dependency).

The table and bar charts in figure 6.5 give the execution time of *across* on all the simulation models of DLX and PERL. The least execution time is seen in *md4* model with branch prediction (*bp* in DLX and *bs* in PERL).

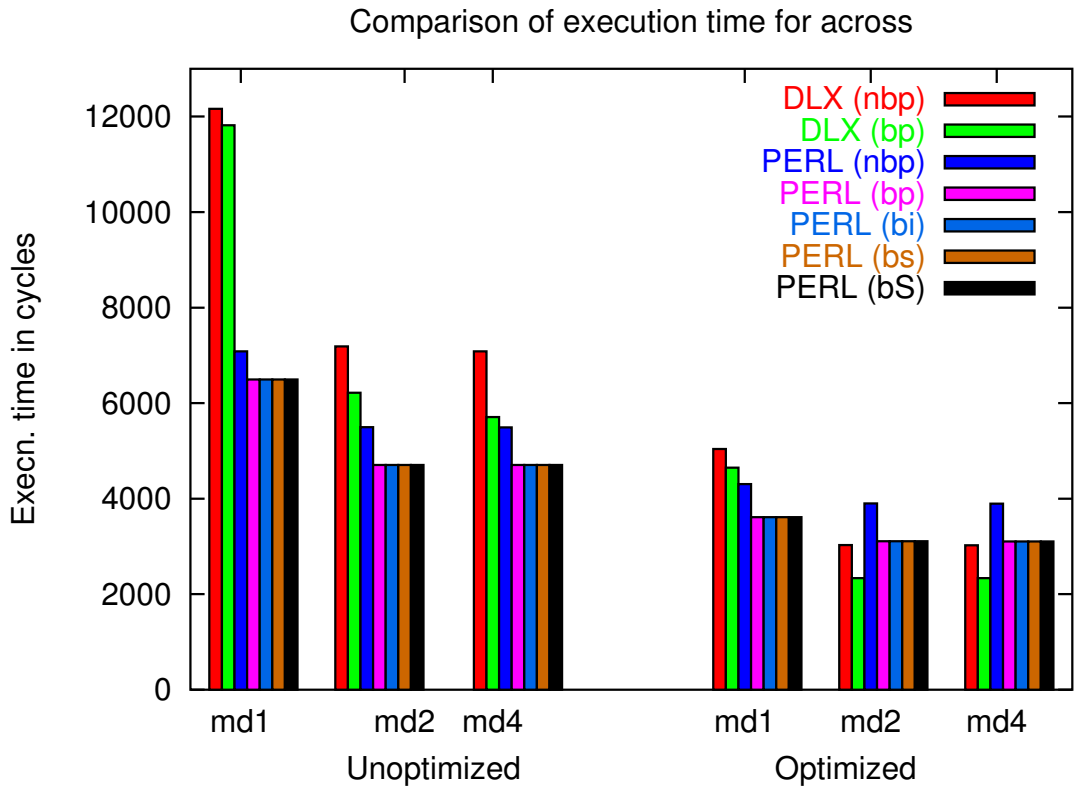
PERL consistently takes fewer cycles to execute the unoptimized code of *across*. However, DLX-*bp* with *md2* and *md4* configurations take smaller execution time to execute the optimized code compared to its PERL counterparts. It can also be observed that the improvement from *md2* to *md4* models when executing optimized code is only a few cycles (1 cycle in DLX and 5 cycles in PERL); this indicates that *across* has a poor ILP. This is indeed true as there are true data dependencies between the statements in the loop.

The performance of various branch prediction schemes used in DLX and PERL is shown in table 6.8. The performance of *bp* is same for both DLX and PERL with a success rate of 98%.

The table 6.9 shows the number of fetch and decode stalls. The number of fetch and decode stalls in DLX are significantly fewer than those in PERL. As seen with all benchmark programs, the number of fetch and decode stall cycles reduce when the branch prediction is used.

The reduction in fetch and decode stalls in DLX is more pronounced than in PERL. In PERL 99% of the fetch stalls are due to filled up instruction queue, while DLX does not have any such stalls.

The average number of instructions processed per cycle is shown in table 6.10.



Config	DLX		PERL				
	nbp	bp	nbp	bp	bi	bs	bS
Unoptimized code							
md1	12162	11820	7084	6493	6493	6493	6493
md2	7186	6219	5495	4707	4707	4707	4707
md4	7086	**5707**	5494	4706	4706	**4706**	4706
Optimized code							
md1	5041	4647	4303	3614	3614	3614	3614
md2	3027	2337	3898	3110	3110	3110	3110
md4	3026	**2336**	3893	3105	3105	**3105**	3105

Figure 6.5: Execution time in cycles for *across*

DLX-*md4*		PERL-*md4*				
nbp	bp	nbp	bp	bi	bs	bS
Fetch stalls in Opt. code of *across* on *md4*						
1816	16	3091	1985	1985	1985	1985
60.01%	0.68%	79.4%	63.93%	63.93%	63.93%	63.93%
Decode stalls in Opt. code of *across* on *md4*						
608	13	2693	2001	2001	2001	2001
20.09%	0.56%	69.18%	64.44%	64.44%	64.44%	64.44%

Table 6.9: Fetch and decode stall cycles in *across* (percentage of total cycles)

The DLX shows a significant improvement in CPC from *md1* to *md4* with the unoptimized code. While PERL shows no improvement in CPC from *md2* to *md4*. However, with the optimized code, both PERL and DLX show no improvement in CPC from *md2* to *md4*. This shows that the average ILP in DLX is about 2. With the optimized code DLX has a CPC of 1 in *md1* model.

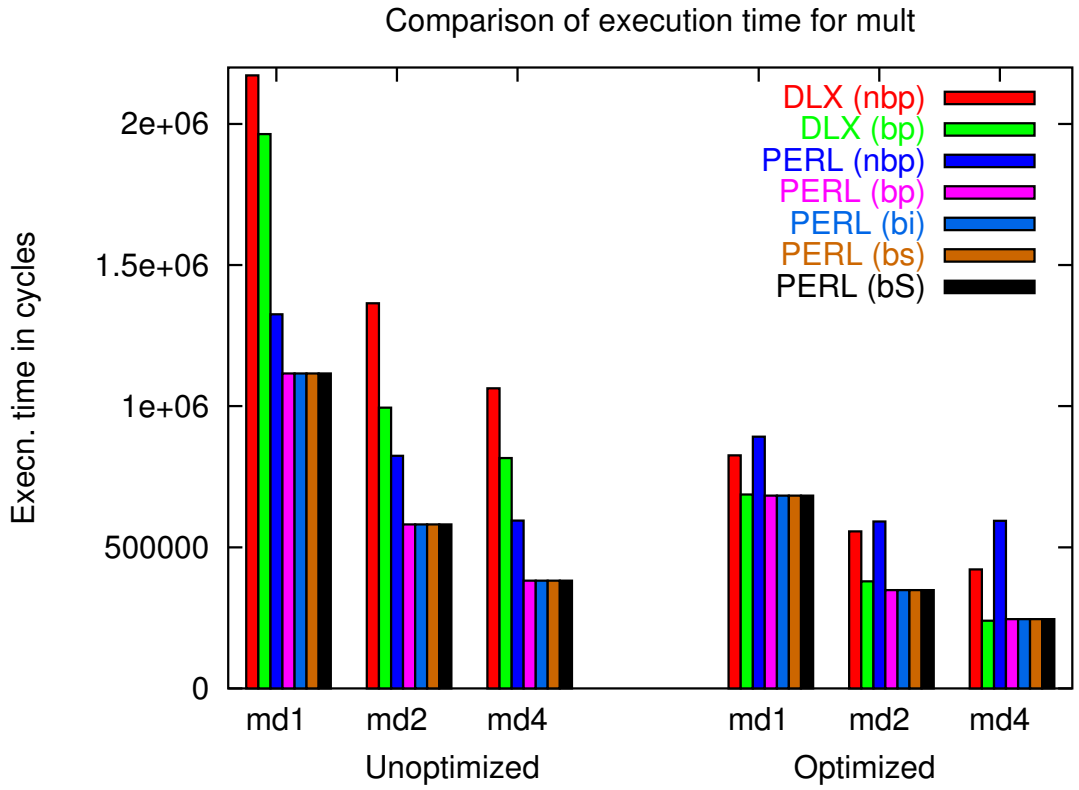
PERL performs better than DLX in all the configurations except in *md2* and *md4* with optimized code. DLX is able to extract more ILP (1.98) than PERL (0.9). Even though DLX executes about 1.65 times more instructions than those in PERL, it has a better execution time as it is able to extract twice the ILP than that found in PERL.

6.4.4 mult benchmark

The mult benchmark initializes two matrices, and computes the product of these two matrices using nested `for` loops of depth six (sub block multiplication). Both matrices are 32x32 in size. The key features in this program are the order in which the elements of the matrices are accessed and variable loop parameters.

The table and bar graphs in figure 6.6 give the execution time of *mult* for both unoptimized and optimized code. The execution time is least in *md4* model with branch prediction (*bp* in DLX and *bs* in PERL).

An interesting feature in figure 6.6 is that with the unoptimized code the speedup



Config	DLX		PERL				
	nbp	bp	nbp	bp	bi	bs	bS
Unoptimized code							
md1	2171865	1963842	1325191	1115923	1115923	1115923	1115923
md2	1364579	994482	824248	581156	581156	581156	581156
md4	1063487	**816146**	594869	381443	381443	**381443**	381443
Optimized code							
md1	825592	687136	892038	682755	682755	682755	682755
md2	556059	379585	591835	348730	348730	348730	348730
md4	421901	**240182**	593528	245272	245272	**245272**	245272

Figure 6.6: Execution time in cycles for *mult*

Model	FPC	DPC	EPC	CPC
Unoptimized Code				
DLX-md1-bp	0.99	0.97	0.96	0.96
PERL-md1-bs	0.93	0.92	0.92	0.91
DLX-md2-bp	1.88	1.84	1.83	1.83
PERL-md2-bs	1.29	1.27	1.27	1.27
DLX-md4-bp	2.04	2.0	1.99	1.99
PERL-md4-bs	1.29	1.27	1.27	1.27
Optimized Code				
DLX-md1-bp	1.0	1.0	1.0	1.0
PERL-md1-bs	0.79	0.78	0.78	0.78
DLX-md2-bp	1.99	1.99	1.98	1.98
PERL-md2-bs	0.92	0.90	0.90	0.90
DLX-md4-bp	2.04	1.99	1.98	1.98
PERL-md4-bs	0.93	0.91	0.90	0.90

Table 6.10: Instructions fetch, decode, issue and commit per cycle (for *across*)

obtained in both DLX and PERL from models *md1* to *md4* is almost linear. However, the speedup from *md2* to *md4* is not as pronounced as from *md1* to *md2*. This gives an indication that there is good amount of ILP available in *mult*. Also for optimized code both PERL and DLX show the same kind of speed up from *md1* to *md2*.

The most important feature while executing the optimized code is that the *md4* model of DLX with *bp* takes marginally smaller time than in the corresponding model of PERL. The reason for this feature is that although PERL still executes fewer instructions than in DLX, the ratio is much smaller as seen in table 6.2. In optimized code DLX executes only about 0.6% more instructions than PERL. While in unoptimized code DLX executes about 73% more instructions than PERL.

The performance of branch prediction schemes used in DLX and PERL is shown in table 6.11. It may be noted that the number of branches executed in DLX and

Program	Metric	DLX-*md4*	PERL-*md4*			
		bp	bp	bi	bs	bS
		D_BR	D_BR	IND_BR	IND_BR	IND_BR
mult	Total #	70857	70857	0	0	0
	Correct	69756	69756	—	—	—
	Succ.rate	98.45%	98.45%	—	—	—

Table 6.11: Success rate of different branch prediction schemes in *mult*

DLX-*md4*		PERL-*md4*				
nbp	bp	nbp	bp	bi	bs	bS
Fetch stalls in Opt. code of *mult* on *md4*						
214737	12373	288453	20332	20332	20332	20332
50.9%	5.15%	58.45%	8.29%	8.29%	8.29%	8.29%
Decode stalls in Opt. code of *mult* on *md4*						
216813	32931	287430	41143	41143	41143	41143
51.39%	13.71%	58.24%	16.77%	16.77%	16.77%	16.77%

Table 6.12: Fetch and decode stall cycles in *mult* (percentage of total cycles)

PERL is identical. The performance of *bp* is same in both DLX and PERL with a success rate of 98%.

The number of fetch and decode stall cycles are presented in table 6.12. Though DLX has fewer fetch and decode stall cycles than those in PERL, it can be observed that the difference is not as big as found in other benchmark programs. The number of fetch stalls due to filled up instruction queue is 95% in PERL and 91% in DLX. The average number of instructions processed per cycle at each stage of the pipeline is shown in table 6.13. Both PERL and DLX achieve good numbers here. An interesting observation is that while executing unoptimized code, DLX show better IPC in *md1* and *md2* where as PERL has better IPC in *md4*. On the other hand, for the optimized code DLX has better IPC in *md1* and *md4*, while PERL has better IPC in *md2*.

Model	FPC	DPC	EPC	CPC
Unoptimized Code				
DLX-md1-bp	0.9	0..89	0.89	0.89
PERL-md1-bs	1.0	1.0	1.0	1.0
DLX-md2-bp	1.99	1.95	1.95	1.94
PERL-md2-bs	1.92	1.91	1.91	1.91
DLX-md4-bp	2.48	2.40	2.38	2.36
PERL-md4-bs	3.01	2.93	2.91	2.91
Optimized Code				
DLX-md1-bp	1.0	0.99	0.99	0.99
PERL-md1-bs	1.0	0.99	0.99	0.99
DLX-md2-bp	1.81	1.8	1.8	1.8
PERL-md2-bs	1.96	1.95	1.94	1.94
DLX-md4-bp	3.01	2.86	2.85	2.84
PERL-md4-bs	2.92	2.78	2.77	2.76

Table 6.13: Instructions fetch, decode, issue and commit per cycle (for *mult*)

6.4.5 ttn benchmark

The *ttn* is a time table scheduler program picked up from our lab exercises. This takes the information from two disk files and hence performs input output. The *ttn* program has some floating point operations.

The table and bar charts in figure 6.7 give the execution time of *ttn* on all the simulation models of DLX and PERL. The results are presented only for unoptimized code. The execution time is the least in *md4* model with branch prediction (*bp* in DLX and *bs* in PERL). The speed up obtained by DLX and PERL from *md1* to *md2* is more significant than that from *md2* to *md4*. PERL takes fewer cycles than DLX to execute *ttn* across all models. Interestingly *md2* model of DLX with *bp* performs

Program	Metric	DLX-*md4*	PERL-*md4*			
		bp	bp	bi	bs	bS
		D_BR	D_BR	IND_BR	IND_BR	IND_BR
ttn	Total #	293049	350165	146587	1456587	1456587
	Correct	255752	308954	108017	1465587	1456389
	Succ.rate	87.27%	88.23%	73.69%	100%	99.99%

Table 6.14: Success rate of different branch prediction schemes in *ttn*

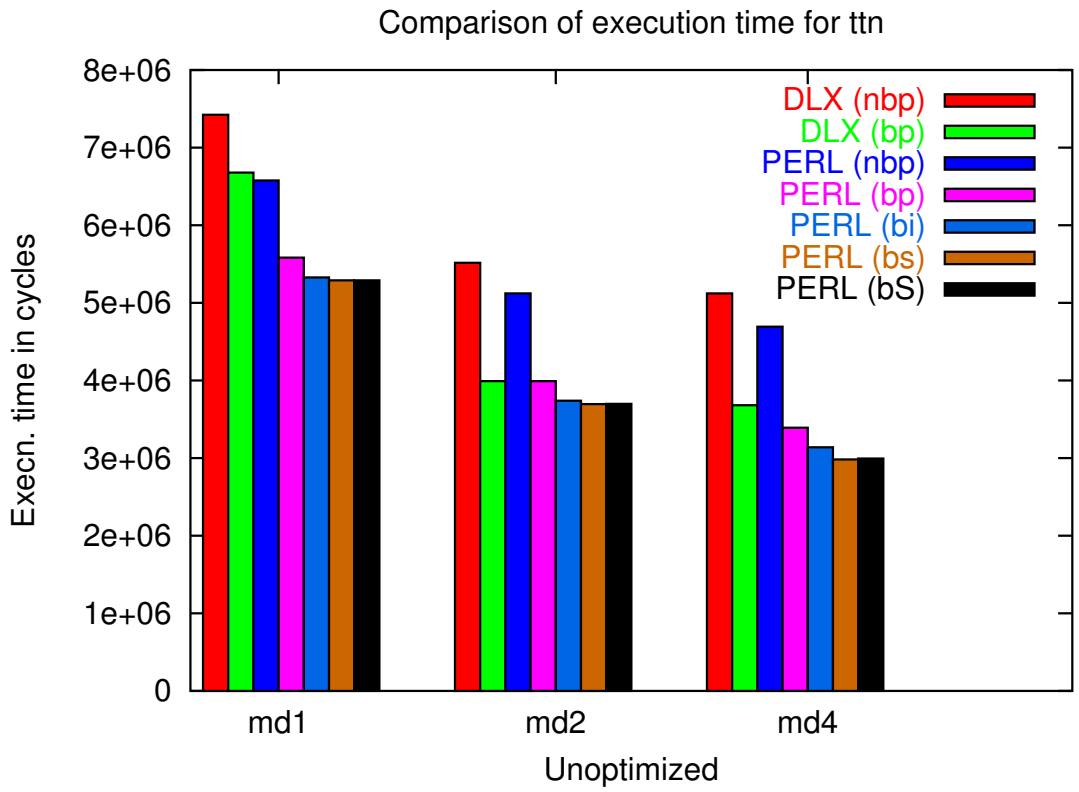
DLX-*md4*		PERL-*md4*				
nbp	bp	nbp	bp	bi	bs	bS
Fetch stalls in Unopt. code of *ttn* on *md4*						
3284030	1537755	3219214	1794590	1499413	1495136	1495983
64.13%	41.78%	68.6%	52.91%	47.79%	50.16%	50.16%
Decode stalls in Unopt. code of *ttn* on *md4*						
1819295	246213	3241181	1882800	1627950	1602065	1602769
35.53%	6.69%	69.07%	55.51%	51.88%	53.74%	53.74%

Table 6.15: Fetch and decode stall cycles in *ttn* (percentage of total cycles)

better than the corresponding model of PERL, however by using *bi*, *bs* and *bS* PERL outperforms DLX.

The performance of various branch prediction schemes used in DLX and PERL is shown in table 6.14. It may be noticed that the number of branches in DLX and PERL is not identical (contrary to what is assumed in the analytical model described in section 3.6).

The table 6.15 gives the number of fetch and decode stalls. The



Config	DLX		PERL				
	nbp	bp	nbp	bp	bi	bs	bS
Unoptimized code							
md1	7848946	6747745	6576652	5581954	5328272	5288858	5289691
md2	5418521	3806608	5121319	3992075	3737833	3697008	3698083
md4	5121131	**3680393**	4692468	3392068	3137703	**2982786**	2994078

Figure 6.7: Execution time in cycles for *ttn*

Model	FPC	DPC	IPC	CPC
Unoptimized Code				
DLX-md1-bp	0.99	0.97	0.95	0.94
PERL-md1-bs	0.75	0.73	0.72	0.69
DLX-md2-bp	1.90	1.81	1.77	1.69
PERL-md2-bs	1.12	1.08	1.05	0.99
DLX-md4-bp	2.20	1.84	1.80	1.74
PERL-md4-bs	1.48	1.39	1.32	1.23

Table 6.16: Instructions fetch, decode, issue and commit per cycle (for *ttn*)

The results show that the improvement in CPC from *md1* to *md2* is significant from 0.94 to 1.69 in DLX, whereas it is 0.69 to 0.99 in case of PERL. The improvement in CPC from *md2* to *md4* is not appreciable, DLX improves from 1.61 to 1.74 while PERL improves from 0.99 to 1.23. The better IPC in DLX is not enough to outperform PERL because DLX executes 70% more number of instructions than those in PERL.

6.4.6 compress benchmark

The *compress* benchmark program is the in-memory version of the standard UNIX compress utility. Files of variable lengths are created, compressed and decompressed. The original file is compared with the compressed/decompressed file. Though, the program is categorized as integer benchmark, *compress* uses a few floating point data, primarily to generate random numbers. The program does not perform any disk access, however, it has few `printf` statements to display the results. The *compress* program requires three input values, which are given as the test data values (supplied with the benchmark).

The bar charts and the table in figure 6.8 give the execution time of *compress* on all the simulation models of DLX and PERL. The results are presented for unoptimized code. The least execution time is seen for *md4* model with branch

Program	Metric	DLX-*md4*	PERL-*md4*			
		bp	bp	bi	bs	bS
		D_BR	D_BR	IND_BR	IND_BR	IND_BR
compress	Total #	249610	214581	48404	48404	48404
	Correct	242370	200872	10904	44144	43932
	Succ.rate	97.1%	97.77%	24.53%	91.19%	81.23%

Table 6.17: Success rate of different branch prediction schemes in *compress*

prediction (*bp* in DLX and *bs* in PERL).

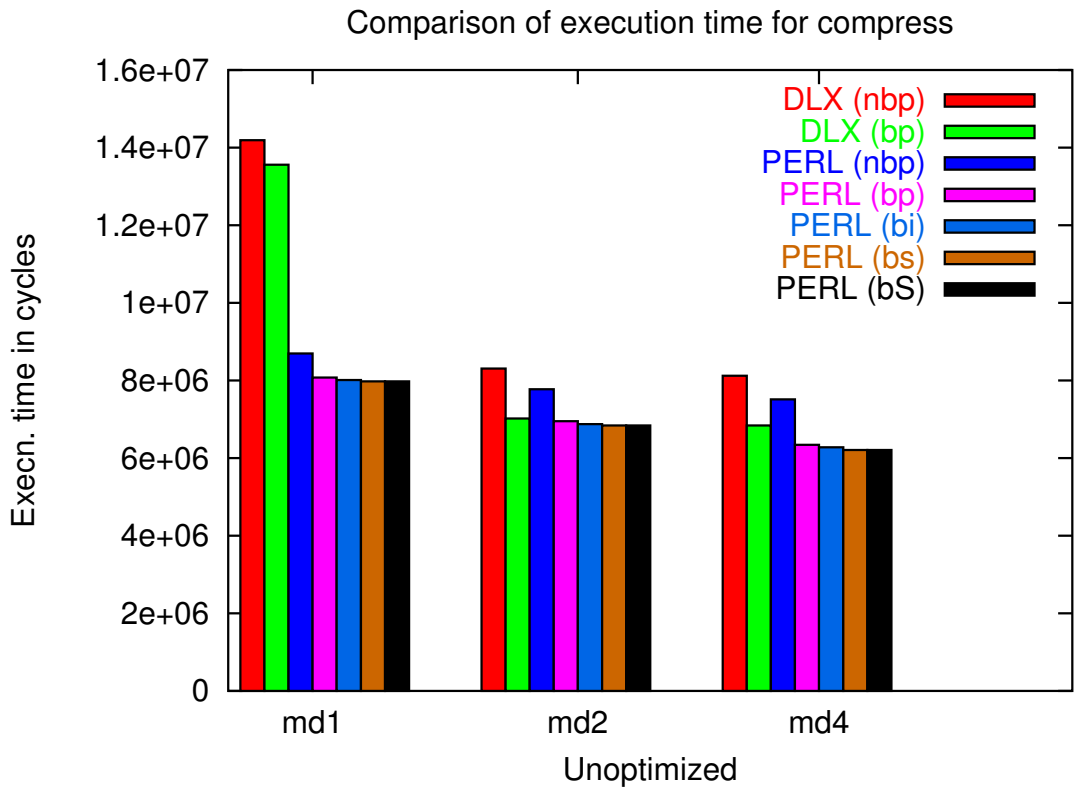
The speed up obtained by DLX and PERL from *md1* to *md2* is significant while the speedup from *md2* to *md4* is marginal. It indicates that the average ILP available in this program is about two for DLX. The results in table 6.19 also show this.

The performance of various branch prediction schemes used in DLX and PERL is shown in table 6.17. It is noticed that the number of branches executed in DLX and PERL is not identical (contrary to what is assumed in the analytical model described in section 3.6). Further, as *compress* is fairly large program, PERL has a few collisions in BTB when indirect branch prediction schemes are used. However, DLX did not encounter any collision in BTB.

The number of fetch and decode stall cycles in DLX and PERL is presented in table 6.18. The reduction in fetch and decode stalls in DLX is more pronounced than in PERL. This is because, while executing *compress*, 92% of the fetch stalls in PERL and about 17% in DLX, are due to filled up instruction queue.

The average number of instructions processed per cycle is shown in table 6.19. The results show that the improvement in CPC from *md1* to *md2* is significant from 0.98 to 1.89 in DLX, while it is from 0.73 to 0.85 in case of PERL. The improvement from *md2* to *md4* is not appreciable, DLX improves from 1.89 to 1.94 while PERL improves from 0.85 to 0.91. This clearly shows that *compress* has a limited ILP.

The results show that PERL performs consistently better than DLX in all models. However, it should be noted that both DLX and PERL codes are unoptimized.



Config	DLX		PERL				
	nbp	bp	nbp	bp	bi	bs	bS
Unoptimized code							
md1	14189482	13562222	8696015	8073333	8010428	7976525	7977138
md2	8308085	7020037	7772519	6953266	6874186	6839400	6840293
md4	8119692	**6838387**	7510513	6342390	6276756	**6209845**	6210838

Figure 6.8: Execution time in cycles for *compress*

DLX-*md4*		PERL-*md4*				
nbp	bp	nbp	bp	bi	bs	bS
Fetch stalls in Unopt. code of *compress* on *md4*						
3892719	421400	5694825	3955507	3826342	3831942	3832714
47.94%	6.16%	75.82%	62.37%	61.71%	61.71%	61.71%
Decode stalls in Unopt. code of *compress* on *md4*						
448449	146581	4838546	3977685	3910684	3877215	3877940
17.84%	2.14%	64.42%	62.72%	62.30%	62.44%	62.44%

Table 6.18: Fetch and decode stall cycles in *compress* (percentage of total cycles)

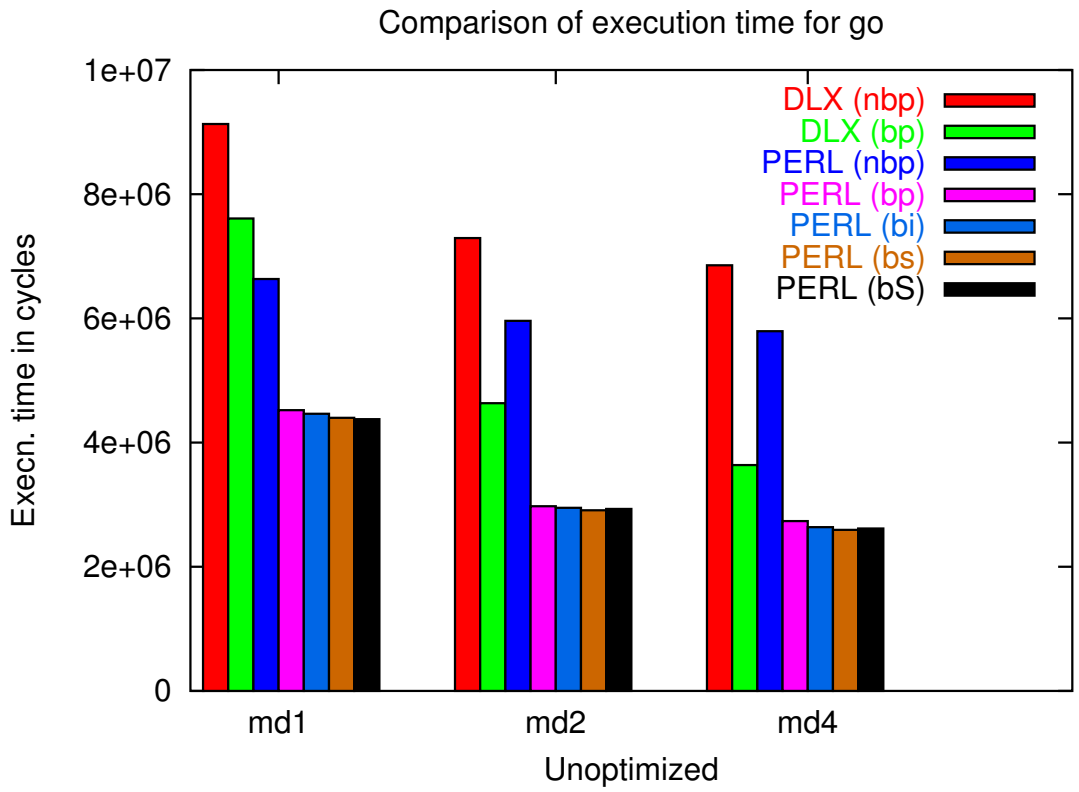
Model	FPC	DPC	EPC	CPC
Unoptimized Code				
DLX-md1-bp	1.0	0.98	0.98	0.98
PERL-md1-bs	0.75	0.74	0.73	0.73
DLX-md-bp	1.93	1.89	1.89	1.89
PERL-md2-bs	0.88	0.86	0.85	0.85
DLX-md4-bp	2.07	1.95	1.95	1.94
PERL-md4-bs	0.95	0.93	0.91	0.91

Table 6.19: Instructions fetch, decode, issue and commit per cycle (for *compress*)

6.4.7 go benchmark

The *go* is go-playing computer program. It is set up to play a single game of Go against itself, with the game record sent to `stdout`. This is an integer program and requires two input values to start the game, which are given the test data values (supplied with the benchmark). We stop the game after five moves. The main feature of this program is the extensive use of different control statements (`if then else`, `switch` etc.).

We present the execution time for *go* program on all the simulation models in figure 6.9. PERL consistently takes fewer cycles to execute the *go* program than the



Config	DLX		PERL				
	nbp	bp	nbp	bp	bi	bs	bS
Unoptimized code							
md1	9131970	7607180	6632591	4523526	4463294	4398139	4376428
md2	7293239	4635033	5960036	2974764	2948325	2910994	2930462
md4	6854394	**3636048**	5793400	2734721	2637416	**2593769**	2614366

Figure 6.9: Execution time in cycles for *go*

Program	Metric	DLX-*md4*	PERL-*md4*			
		bp	bp	bi	bs	bS
		D_BR	D_BR	IND_BR	IND_BR	IND_BR
go	Total #	605581	684598	110419	110419	110419
	Correct	554868	617204	67488	101751	95313
	Succ.rate	90.59%	90.16%	61.12%	92.14%	86.32%

Table 6.20: Success rate of different branch prediction schemes in *go*

DLX-*md4*		PERL-*md4*				
nbp	bp	nbp	bp	bi	bs	bS
Fetch stalls in Unopt. code of *go* on *md4*						
4639382	572975	4350854	1248127	1117122	1012089	1296726
67.68%	15.76%	75.10%	45.64%	42.40%	39.02%	40.12%
Decode stalls in Unopt. code of *go* on *md4*						
4712905	786841	4364427	1425337	1334687	1283916	1296726
68.76%	21.64%	75.33%	52.12%	50.60%	49.50%	49.60%

Table 6.21: Fetch and decode stall cycles in *go* (percentage of total cycles)

DLX across all models. The least execution is seen in *md4* with branch prediction (*bp* in DLX and *bs* in PERL). The speedup obtained from *md1* to *md2* is significant, where as only a marginal improvement is seen in speedup from *md2* to *md4* in both DLX and PERL.

Table 6.20 gives the performance of all the branch prediction schemes used in DLX and PERL. The *go* program contains indirect branches (to implement the C `switch` construct), these are not predicted in PERL (only indirect branches which use base addressing are predicted).

We present the number of fetch and decode stall cycles in DLX and PERL in table 6.21. DLX has fewer fetch and decode stalls than those in PERL as in case of other benchmark programs.

The average number of instructions processed per cycle is presented in table 6.22. The *go* program exhibits a CPC of 2 in DLX on *md4*, while on the same model PERL

Model	FPC	DPC	IPC	CPC
Unoptimized Code				
DLX-md1-bp	0.99	0.96	0.95	0.95
PERL-md1-bs	0.98	0.95	0.92	0.91
DLX-md2-bp	1.78	1.70	1.63	1.56
PERL-md2-bs	1.71	1.62	1.41	1.37
DLX-md4-bp	2.65	2.36	2.20	1.99
PERL-md4-bs	2.10	1.98	1.78	1.54

Table 6.22: Instructions fetch, decode, issue and commit per cycle (for *go*)

has a CPC of 1.54.

The results show that PERL performs better than the DLX in all models. However, the results presented are for the unoptimized code.

6.5 Other Issues

The results presented in the previous section show that PERL performs better than DLX for most of the benchmarks. For many programs PERL performs better even against the optimized code of DLX.

There may be many reasons behind PERL to perform better than DLX, but some of the important ones are listed below.

1. Fewer instructions.
2. Operand forwarding / Operand renaming.
3. SP/FP accesses.

We have already seen that PERL executes fewer instructions (figure 6.2). We present the results with respect to the other two points here.

Program	PERL-md4-bs					
	Tot. Acc	Avg/ins	Tot. Rd	Avg/ins	Tot. Wr	Avg/ins
perm	21316	2.03	11745	1.12	9571	0.91
relax	1986332	1.64	883257	0.73	1103075	0.91
across	5583	1.99	2981	1.06	2602	0.93
mult	1194162	1.76	588702	0.87	605460	0.89
ttn	6570941	1.79	3558522	0.97	3012419	0.82
compress	14441838	2.46	9014709	1.54	5427129	0.93
go	8699505	2.14	5300629	1.31	3398876	0.84

Table 6.23: Memory references in PERL

6.5.1 Operand Forwarding / Operand Renaming

An interesting related issue in PERL is the number of memory accesses made during the program execution.

Each PERL instruction performs six memory accesses in the worst case. The worst case occurs when all operands of the instruction are specified using indirect addressing. However, in programs many operands are specified using direct and immediate addressing as well. In addition, unconditional jump instructions use only two operands. Hence, on the average the number of memory requests per instruction during execution of the program is expected to be smaller than six.

We present the actual number of memory references made by PERL in table 6.23. These include the memory references made due to speculative execution.

Interestingly the average number of memory accesses made per instruction is very low. This is because many of the addresses and operands required during execution are obtained from data forwarding/operand renaming.

PERL instructions are decoded in two stages, namely, address generation and operand access as shown in figure 4.2 and explained in section 4.2.1. The effective address is computed in the address generation stage for all operands that are specified using indirect or base addressing modes. The actual operand value is fetched in the operand access stage.

In superscalar PERL, all data dependencies among the instructions are resolved

before the instruction is decoded and placed in the instruction window. To improve the ILP, PERL uses operand renaming and address/operand forwarding. The decode unit places the decoded instructions in the reorder buffer. The destination addresses of all the instructions are then renamed to the reorder buffer entry number (thereby eliminating the anti-dependencies). New instructions coming into the pipeline may find their operands in the reorder buffer.

In the address generation stage, memory accesses need to be made to get the value of the base. The base addresses for the indirect operands (destination and source) are searched in the reorder buffer to get the value. A match indicates that a previously decoded instruction in the pipeline will be providing the value. If a match is found then the effective address is picked up from the reorder buffer entry provided the entry was valid. In case a match is found and the corresponding instruction is yet to write-back the data (i.e. the reorder buffer entry is invalid), the processor stalls the current instruction. If the match was searched for the source operand, the decoding is continued for further instructions. However, if the match was searched for destination operand, the processor stalls and does not decode further instructions. These stalls are named decode stalls. The decoding is resumed later when the instruction owning the address is executed completely. However, if there was no match, the processor performs a memory read to obtain the address. Thus, every successful address search avoids one memory access to obtain the effective address of an indirect operand.

Similarly, in the operand access stage, the operands are read from the memory. For this the addresses are searched in the reorder buffer and a similar action is performed. Thus, memory accesses are avoided if the corresponding value was found in the reorder buffer.

We present the number of addresses and operand searches made, and the success rate in table 6.24. The success rate of address search ranges from 20% to 100%. It can also be observed that the success rate of finding the operand in reorder buffer ranges from 25% to 65%. As each successful search avoids a memory access, the actual number of memory accesses comes down as shown in table 6.23.

Program	**Address Search**		**Operand Search**	
	# of srch	Succ. rate	# of srch	Succ. rate
perm	8840	65.18%	10337	27.79%
relax	492104	100%	1885536	53.16%
across	1595	93.73%	3791	24.00%
mult	378476	35.18%	983743	65.09%
ttn	2397090	20.39%	2349502	61.06%
compress	6766680	31.31%	5640950	25.41%
go	5034192	28.26%	9345672	66.48%

Table 6.24: Success rate of finding addresses and operands in reorder buffers

6.5.2 SP/FP Accesses

Programs make frequent access to local variables during execution, which are allocated on the stack by the compilers. The frame and stack pointer registers are frequently used to access the local variables. PERL is expected to cache SP and FP in L0 cache to give almost 100% hit. The number of SP/FP accesses and total memory accesses are shown in table 6.25. It can be observed that in case of some programs the combined SP and FP accesses are about 30% to 38% of total memory accesses. Programs *relax*, *across* and *mult* implement one function each that uses global arrays and hence have very few accesses to SP and FP.

The high percentage of SP and FP accesses justifies our assumption that the access to these will yield a hit ratio of almost 100% in L0 cache.

The execution of programs on the simulator has shown that PERL perform better than DLX for most of the programs. With the exception of *mult* and *across*, where the optimized code for DLX performs better than the optimized code for PERL, the PERL code consistently runs faster than the DLX. It may be noted that the optimizations in PERL do not include any machine specific optimizations.

Shown as percentage of Total memory access						
Program	SP/FP. Acc	SP. Rd	SP. Wr	Mem. Acc	Mem. Rd	Mem. Wr
perm	7509	5309	2200	13807	6436	7371
	35.23%	45.20%	22.99%	64.77%	54.80%	77.01%
relax	1158	1135	23	1985174	882122	1103052
	0.06%	0.13%	0.00%	99.94%	99.87%	100.00%
across	14	11	3	5569	2970	2599
	0.25%	0.37%	0.12%	99.75%	99.63%	99.88%
mult	245328	245325	3	948834	343377	605457
	20.54%	41.67%	0.00%	79.46%	58.33%	100.00%
ttn	2554953	2255018	299935	4015988	1303504	2712484
	38.88%	63.37%	9.96%	61.12%	36.63%	90.04%
compress	4884016	4761274	122742	9557822	4253435	5304387
	33.82%	52.82%	2.26%	66.18%	47.18%	97.74%
go	3448334	2992205	456129	5251171	2308424	2942747
	39.64%	56.45%	13.42%	60.36%	43.55%	86.58%

Table 6.25: Accesses to SP/FP and to other memory locations in PERL

Chapter 7

Analysis of Memory System

The performance of memory subsystem plays a very significant role in the overall performance of the machine. PERL being a purely memory-to-memory architecture, the performance of memory subsystem is even more significant to the performance of the processor. There are several studies [46, 51, 66, 104, 105], discussing the performance of memory subsystem for RISC processors including that for DLX. However, there are no such studies for a memory-to-memory processor like PERL.

We analyze the performance of cache in both DLX and PERL. The study is based on the memory trace obtained during simulation. Traces are collected for each of the benchmark programs on models *md4* with branch prediction scheme *bp* in DLX and *bs* in PERL. We perform simulations to assess the performance of both instruction and data caches. It is generally known that the data access and instruction access patterns of programs vary differently. Processors of today therefore have a split instruction and data caches at L1 cache hierarchy.

We present the data and code sizes for the benchmark programs in table 7.1 and the total number of instructions fetched for DLX and PERL in table 7.2. These figures are important for analyzing the cache simulation results. The *relax* benchmark has no global variables and the single array in the program is declared as local variable and allocated onto stack, hence its data size is one.

Program	Code Size in bytes		Data Size
	DLX	PERL	in bytes
perm	1232	3280	32
relax	1276	2240	1
across	1028	2224	2000
mult	1312	2704	12288
ttn	21416	42496	70579
compress	12476	23456	44111823
go	499064	1032160	565717

Table 7.1: Code and data sizes of programs in DLX and PERL

Program	DLX-*md4-bp*	PERL-*md4-bs*
perm	21190	17552
relax	1745262	1248908
across	4783	2877
mult	723617	716728
ttn	8117721	4430877
compress	14196657	5899370
go	10060599	5446915

Table 7.2: Instruction fetch count of all programs in DLX and PERL

7.1 Cache Hierarchy in PERL and DLX

The cache hierarchy in DLX and PERL is similar to the one shown in figure 3.3. PERL has a small number of registers mapped on to memory locations. We expect it to cache the most frequently used memory locations. We treat this as L0 cache in our discussion. The likely candidates to occupy these locations are SP, FP, local variables and temporary variables. The access time of this is assumed to be one clock (equal to that of registers). The L0 cache is capable of serving multiple (eight in our model) memory requests in a single cycle in the same way as registers. All requests in PERL first go to L0 as it is the highest level in the memory hierarchy. The misses in L0 cache is passed on to L1 cache. The way in which accesses to L1

and beyond are processed is similar in both DLX and PERL.

It is important to model L0 cache to assess the performance of PERL properly. We modeled L0 cache to contain 16 registers with 8 ports. Registers in some processors today have more than 8 ports. DLX has a register set with 32 registers, the benefit of which is extracted by compiler and superscalar execution (in the simulator).

Some of the common parameters that are used in our simulations are the following.

- L0 cache is present only in PERL. It has 16 registers of 32-bits, fully associative and has 8 ports.
- L1 is a split cache.
- L1 instruction cache is 8 KB, direct mapped and has one port.
- L1 data cache is 8 KB, 32 byte block size, 4 interleaved ports and employs write back policy.
- L1 miss penalty is 4 cycles.
- The cache size of L2 is 256 KB, as this is the minimum size of L2 cache in most current processors.
- L2 is a unified, four-way set associative , write back cache with miss penalty of 13 cycles.

7.2 Instruction Cache

PERL fetches fewer instructions than DLX. However, each PERL instruction is four times (128 bits) the size of a DLX instruction (32 bits). We study the performance of instruction cache by performing two sets of simulation.

A bigger cache block size in instruction cache is likely to improve the performance of PERL more than DLX. A bigger instruction cache is also likely to benefit

PERL more than the DLX. We study these two issues through the following set of simulations.

- **Constant cache size with varying block size**: We performed three simulations on each trace. The size of L1 I-cache (8KB) and L2 cache (256KB) are kept constant. The simulation are conducted for block sizes of 32, 64 and 128 bytes respectively (in both L1 and L2). The associativity of L1 I-cache and L2 cache is kept at 1 and 4 respectively and only one port is provided.

- **Varying Cache size**: The number of blocks in L1 I-cache is kept constant (256). The simulations are conducted for block sizes of 32, 64 and 128 bytes. In effect we study the performance of I-cache for sizes 8 KB, 16 KB and 32 KB respectively. The size of L2 cache is kept constant (256KB), the block size is varied to match the size of the L1 I-cache. The associativity of L1 I-cache and L2 cache is kept at 1 and 4 respectively.

7.2.1 Varying Block Size

The significance of this set of simulation is to observe the effect of larger block size in reducing the number of instruction cache misses.

The performance of L1 I-Cache over this set of simulations is presented in table 7.3. The results show that the number of misses reduce as the block size increases (even though the total cache size is kept constant). The reduction in number of misses is more pronounced in PERL than in DLX across all programs. This is expected as the size of a PERL instruction is larger than that of a DLX instruction. Thus larger cache block size brings in and holds more instructions per block for future use. A bigger block size therefore benefits PERL more than DLX.

The performance of L2 cache is presented in the table 7.4. The results here also show the same trend with the reduction in the number of misses being more pronounced in PERL than in DLX. It can also be observed that all the L1 I-cache misses are compulsory misses except in the case of *ttn* and *compress*.

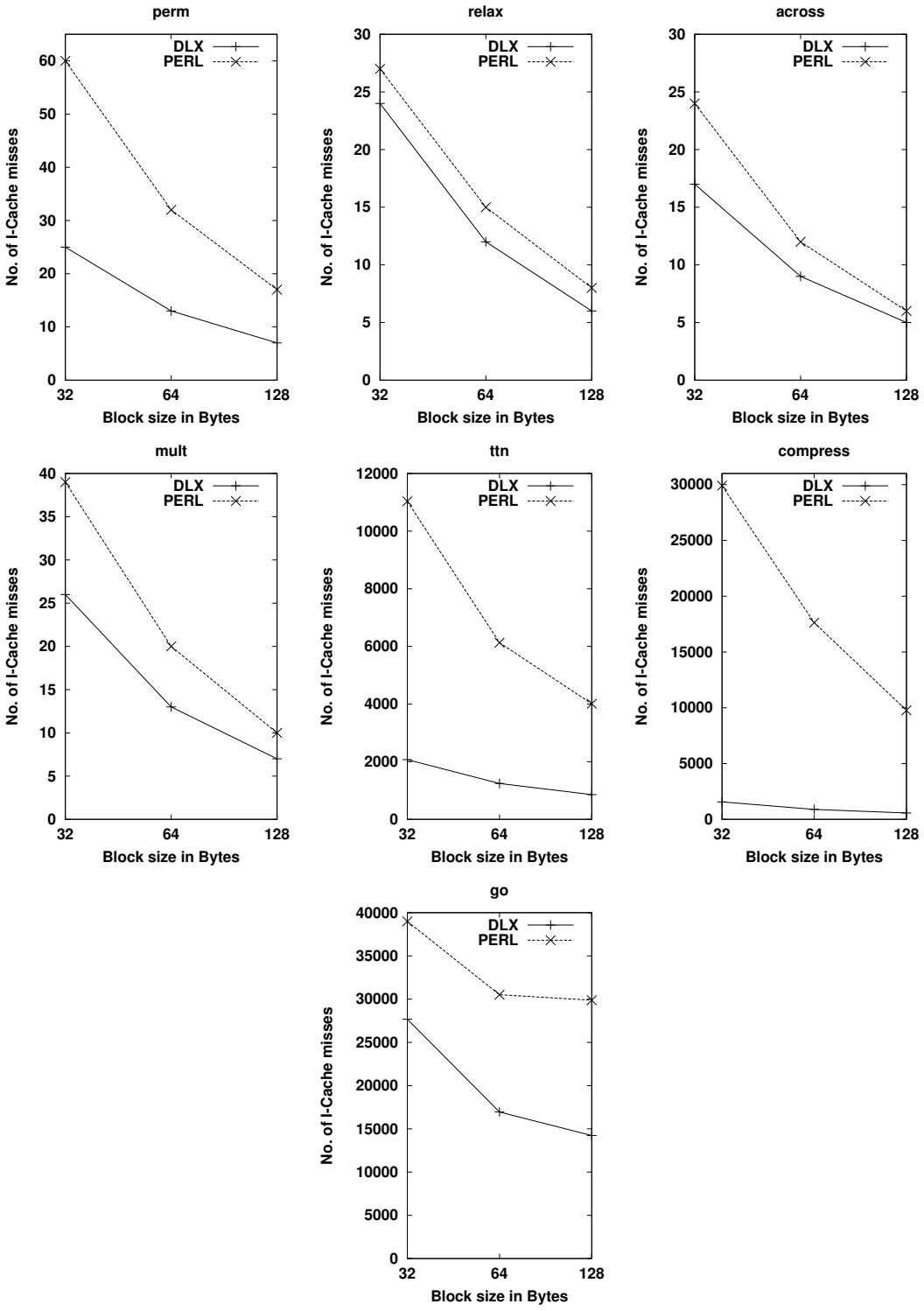


Figure 7.1: L1 I-cache misses with increasing block size

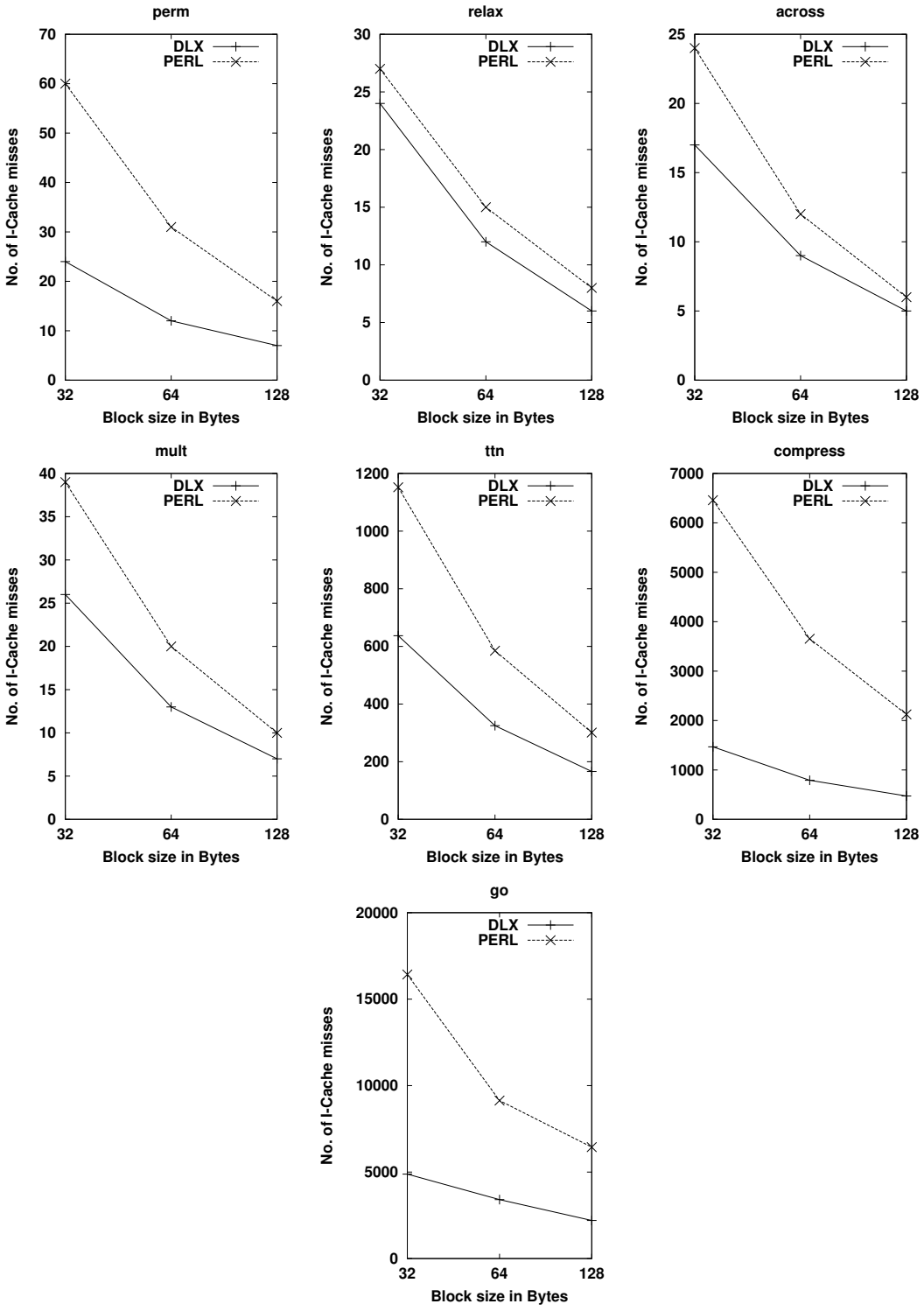


Figure 7.2: L2 I-fetch misses with increasing block size

Program	machine	# Ifetch	32 bytes	64 bytes	128 bytes
			# of imiss	# of imiss	# of imiss
perm	DLX	21190	25	13	7
	PERL	17552	60	32	17
relax	DLX	1745262	24	12	6
	PERL	1248908	27	15	8
across	DLX	4783	17	9	5
	PERL	2877	24	12	6
mult	DLX	723617	26	13	7
	PERL	716728	39	20	10
ttn	DLX	8117721	2070	1244	858
	PERL	4430877	11033	6129	4011
compress	DLX	14196657	1566	893	572
	PERL	5899370	29908	17627	9766
go	DLX	10060599	27681	16949	14226
	PERL	5446915	38989	30508	29874

Table 7.3: L1 I-cache misses with varying block size

Program	Machine	32 bytes		64 bytes		128 bytes	
		# Acc	# imiss	# Acc	# imiss	# Acc	# imiss
perm	DLX	25	24	13	12	7	7
	PERL	60	60	32	31	17	16
relax	DLX	24	24	12	12	6	6
	PERL	27	27	15	15	8	8
across	DLX	17	17	9	9	5	5
	PERL	24	24	12	12	6	6
mult	DLX	26	26	13	13	7	7
	PERL	39	39	20	20	10	10
ttn	DLX	2070	637	1244	325	858	166
	PERL	11033	1152	6129	585	4011	301
compress	DLX	1566	1467	893	791	572	473
	PERL	29908	6460	17627	3655	9766	2125
go	DLX	27681	4882	16949	3409	14226	2203
	PERL	38989	16424	30508	9134	29874	6442

Table 7.4: L2 I-fetch misses for varying L1 I-cache block size

7.2.2 Varying Cache Size

In these set of simulations, the number of sets in L1 I-cache is maintained constant (256). The simulations are conducted for block sizes of 32, 64 and 128 bytes (total cache size equal to 8KB, 16KB and 32KB respectively). The size of L2 cache is however kept constant at 256KB. The significance of this set is to show that PERL responds to bigger instruction caches more favorably than the DLX.

The results for L1 I-cache for this set of simulation are presented in table 7.5. The results substantiate our expectation that instruction cache misses reduce more drastically in PERL with large block size and larger caches. The misses also come down in DLX, but the reduction is much more pronounced in PERL. While all programs show the same trend, the reduction is more dramatic in *ttn* and *compress* than compared to that in *relax*.

The results for L2 cache is presented in table 7.6. Here also the reduction in the number of misses is more pronounced in PERL than in DLX. It may also be noted that the number of L2 accesses reduce as the size of I-cache in L1 increases. Again it is observed that all the L1 misses are compulsory misses except in *ttn* and *compress*.

7.3 Data Cache

From the results obtained, we observe the following characteristics in the memory access pattern of PERL.

- PERL is a memory-to-memory processor and hence generates more memory accesses than DLX.
- Since PERL takes smaller time to execute, the number of memory requests per cycle are generally more than that in DLX.

To be able to meet such memory access requirements, PERL requires an efficient cache system. We present the results of the cache subsystem of cache in both DLX and PERL. While describing the results we consider only the *md4* models of DLX-*bp* and PERL-*bs* which gave the best performance over all benchmarks.

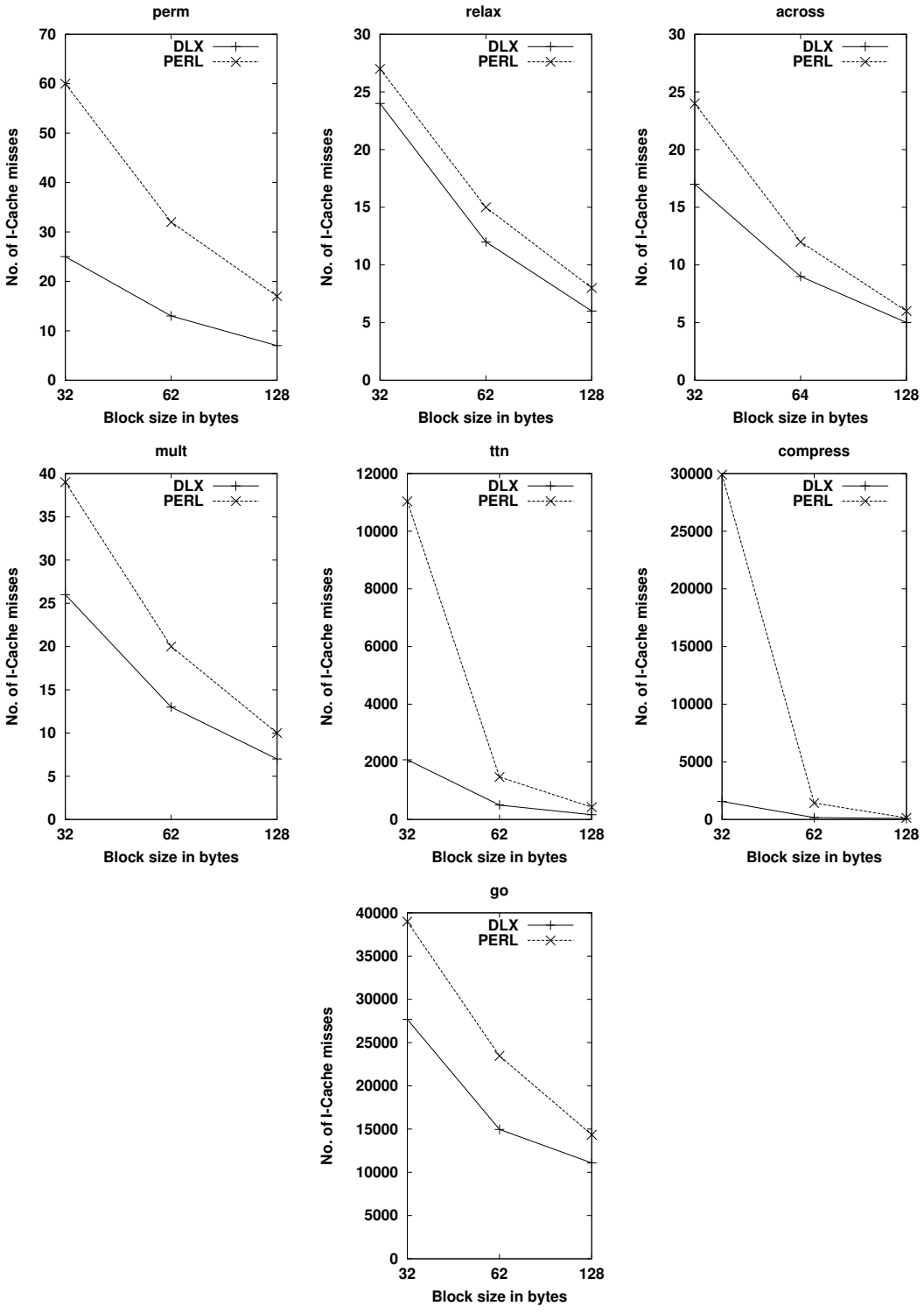


Figure 7.3: Effect of L1 I-cache size on number of L1 I-cache misses

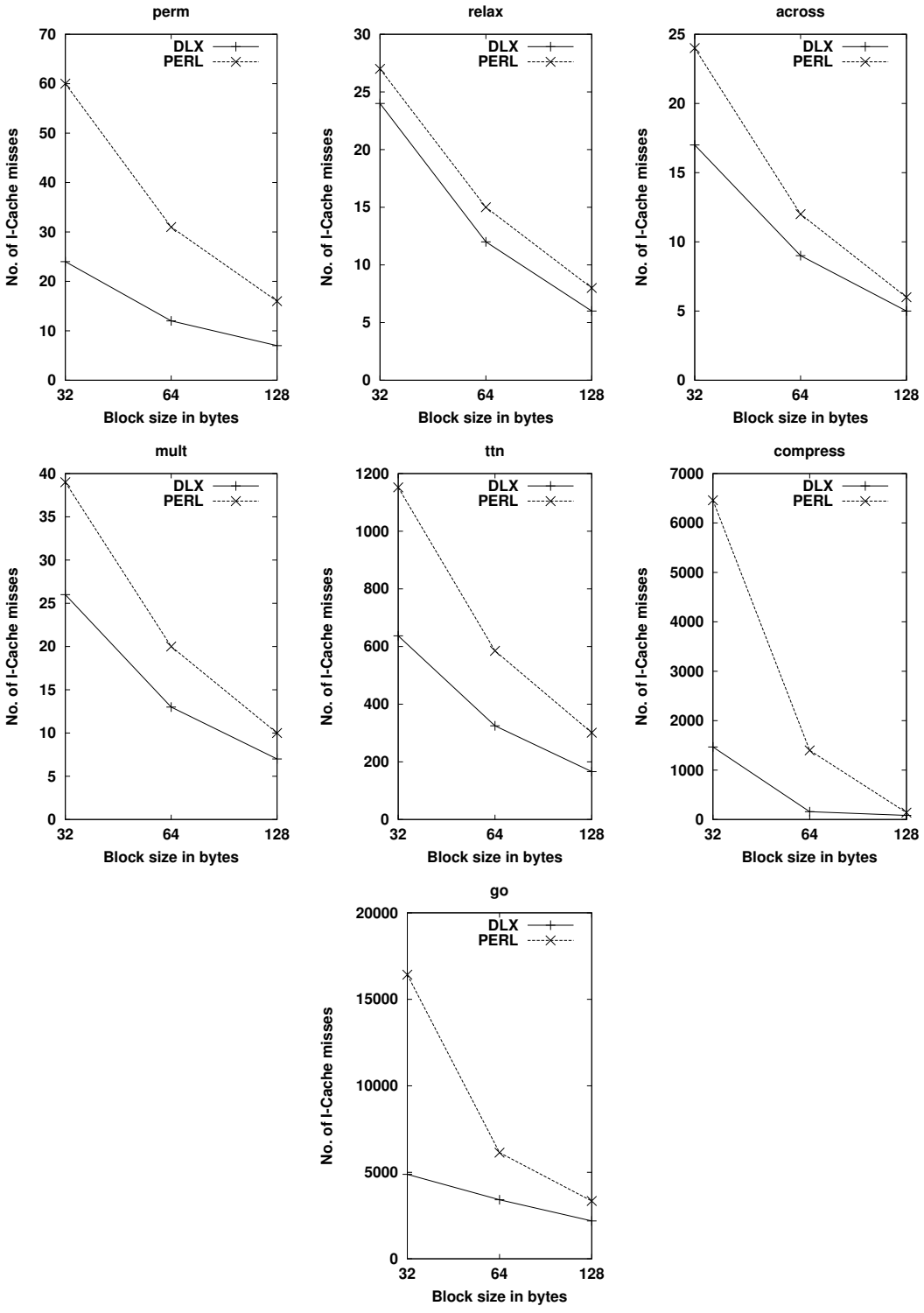


Figure 7.4: Impact of L1 I-cache size on number of L2 I-fetch misses

Program	machine	#Ifetch	32-8KB	64-16KB	128-32KB
			# of imiss	# of imiss	# of imiss
perm	DLX	21190	25	13	7
	PERL	17552	60	32	17
relax	DLX	1745262	24	12	6
	PERL	1248908	27	15	8
across	DLX	4783	17	9	5
	PERL	2877	24	12	6
mult	DLX	723617	26	13	7
	PERL	716728	39	20	10
ttn	DLX	8117721	2070	499	166
	PERL	4430877	11033	1474	429
compress	DLX	14196657	1566	163	85
	PERL	5899370	29908	1426	143
go	DLX	10060599	27681	14955	11088
	PERL	5446915	38989	23454	14345

Table 7.5: L1 I-cache misses with varying L1 I-cache size

Program	Machine	32-8KB		64-16KB		128-32KB	
		# Acc	# imiss	# Acc	# imiss	# Acc	# imiss
perm	DLX	25	24	13	12	7	7
	PERL	60	60	32	31	17	16
relax	DLX	24	24	12	12	6	6
	PERL	27	27	15	15	8	8
across	DLX	17	17	9	9	5	5
	PERL	24	24	12	12	6	6
mult	DLX	26	26	13	13	7	7
	PERL	39	39	20	20	10	10
ttn	DLX	2070	637	499	325	166	166
	PERL	11033	1152	1480	585	430	301
compress	DLX	1566	1467	163	158	85	83
	PERL	29908	6460	1426	1399	143	140
go	DLX	27681	4882	14955	3412	11088	2187
	PERL	38989	16424	23454	6134	14345	3342

Table 7.6: L2 I-fetch misses with increasing L1 I-cache size

7.3.1 Data-Cache Hierarchy

The data cache hierarchy in PERL includes an L0 cache, which are a small number of registers mapped on to memory. The DLX, being a load-store machine has the register set, the benefit of which is statically extracted by the compiler by means of register allocation.

There are many cache enhancement schemes available to improve the cache performance. We consider two of them namely, *multi-port cache* and *load-all-wide* technique. Multi-port caches serve multiple accesses in a single cycle. Where as the load-all-wide technique serves multiple misses to the same cache block simultaneously. Both these techniques help in serving multiple simultaneous memory accesses and hence will benefit both DLX and PERL processors.

The L1 data cache is common to both DLX and PERL. Both DLX and PERL issue multiple memory requests in a single cycle. Therefore, we provide 4 interleaved ports in L1 cache. The other parameters of L1 cache is as given in section 7.1.

7.3.2 Evaluation Methodology

The least execution times are obtained in *md4-bp* of DLX and *md4-bs* of PERL. As these configurations are superscalar and use efficient branch prediction schemes, the memory requirement for these configurations are also the maximum. The memory traces were collected during the execution of programs on simulators configured as *md4-bp* for *dlxsim* and *md4-bs* for *perlsim*. These traces are used to analyze the performance of the cache subsystem.

The performance metrics that are important for our study are miss rate, extra cycles required to serve the misses and bank clashes. Bank clashes are important as they affect the time required to serve multiple simultaneous hits to the same cache bank.

7.3.3 Impact of L0 Cache

The L0 cache is the first level of cache hierarchy in PERL. It consists of 16 registers mapped to memory locations and is expected to cache the most frequently accessed

data. We present the number of misses in L0 cache in table 7.7. We also present the number of write backs from L0 cache in table 7.8. Even with 16 registers L0 cache captures substantial number of hits.

The L0 cache miss rate is the lowest in *ttn* (3.36%) and highest in *mult* (53.95%). The misses out of L0 cache are served by L1 cache and therefore these are significantly reduced. We present the number of accesses that go to L1 cache in DLX and PERL in table 7.9. Interestingly in case of *compress*, *relax* and *ttn* programs fewer accesses are sent to L1 cache in PERL than the number of of Load/Store instructions executed by DLX.

This observation also indicates the effectiveness of dynamic register allocation in PERL as compared to static allocation by the compiler in DLX.

7.3.4 Performance of L1 and L2 Cache

All the parameters of L1 data cache and L2 cache are fixed except the associativity. The performance of L1 and L2 cache are evaluated for 1-way, 2-way and 4-way set associativities. The size of L1 data cache and L2 cache are kept at 8KB and 256KB respectively.

We present the read, write and total miss ratios in L1 cache in table 7.10. The miss ratios decrease as the associativity is increased from 1 to 4 for all programs except *ttn*. We present the variations in the number of read, write and total misses in L1 cache in figure 7.5.

The misses in L2 cache for programs *perm*, *across*, *mult* are constant and comprise of only the compulsory misses in DLX. Except *mult*, the other two programs show the same behavior in PERL. We present the number of read, write and I-fetch misses from L2 cache in table 7.11 for both PERL and DLX and the variations in the number of misses in L2 cache in figure 7.6.

7.3.5 Impact of Load-all-wide Technique

We present the number of misses served by load-all-wide technique in table 7.12. As clear from this table, the load-all-wide scheme benefits PERL more than DLX

Metric	# of Acc	# of Misses	miss_rate
perm			
Reads	8741	2100	24.03%
Writes	9571	2440	25.49%
Total	18312	4540	24.79%
relax			
Reads	883256	323622	36.64%
Writes	1103075	43197	3.92%
Total	1986331	366819	18.47%
across			
Reads	2780	974	35.04%
Writes	2602	807	31.02 %
Total	5382	1781	33.10%
mult			
Reads	479889	325572	67.84%
Writes	605460	259982	42.94%
Total	1085349	585554	53.95%
ttn			
Reads	2773079	104834	3.78%
Writes	3012418	89742	2.97%
Total	5785497	194576	3.36%
compress			
Reads	6529918	356173	5.45%
Writes	5427129	2132082	39.29%
Total	11957047	2488255	20.81%
go			
Reads	4894793	587952	12.01%
Writes	2694921	853914	31.68%
Total	7589714	1441866	19.0%

Table 7.7: Performance of L0 cache with 16 registers in PERL

Program	# of Write backs
perm	3939
relax	108968
across	902
mult	260040
ttn	109397
compress	2300993
go	790450

Table 7.8: Number of write backs from L0 cache

Program	**PERL**			**DLX**		
	Tot. Acc	Tot. Rd	Tot. Wr	Tot. Acc	Tot. Rd	Tot. Wr
perm	8479	4540	3939	8113	3538	4575
relax	475787	366819	108968	488569	382700	105869
across	2683	1781	902	1515	604	911
mult	845594	585554	260040	133158	98319	34839
ttn	303973	194576	109397	1647047	1179578	467469
compress	4789248	2488255	2300993	7007185	4556628	2450557
go	2232316	1441866	790450	2088626	1548650	539976

Table 7.9: Memory references to L1 data cache in DLX and PERL (Total Read include the read and write misses from L0 cache since L0 cache is writeback cache and hence requires a read as well)

Assoc	DLX			PERL		
	Reads	Writes	Total	Reads	Writes	Total
perm						
# of Acc	3528	4575	8113	4540	3939	8479
1-way (m)	0.03%	0.24%	0.15%	0.26%	0%	0.14%
2-way (m)	0.03%	0.24%	0.15%	0.26%	0%	0.14%
4-way (m)	0.03%	0.24%	0.15%	0.26%	0%	0.14%
relax						
# of Acc	382700	105869	488569	366819	108968	475787
1-way (m)	3.27%	1.18%	2.82%	3.89%	0.14%	3.03%
2-way (m)	3.27%	1.18%	2.82%	3.75%	0%	2.89%
4-way (m)	3.27%	1.18%	2.82%	3.75%	0%	2.89%
across						
# of Acc	604	911	1515	1781	902	2683
1-way (m)	0%	7.24%	4.36%	4.38%	0%	2.91%
2-way (m)	0%	7.24%	4.36%	4.38%	0%	2.91%
4-way (m)	0%	7.24%	4.36%	4.38%	0%	2.91%
mult						
# of Acc	98319	34839	133158	585554	260040	845594
1-way (m)	0.53	0.75%	0.59%	3.04%	3.47%	3.16%
2-way (m)	0%	0.75%	0.2%	0.34%	0.15%	0.28%
4-way (m)	0%	0.75%	0.2%	0.09%	0%	0%
ttn						
# of Acc	1179578	467469	1647047	194576	109327	303973
1-way (m)	0.12%	0.04%	0.09%	1.14%	0.07%	0.76%
2-way (m)	0.11%	0.02%	0.08%	0.84%	0%	0.53%
4-way (m)	0.14%	0.02%	0.11%	1.15%	0%	0.74%
compress						
# of Acc	4556628	2450557	7007185	2488255	2300993	4789348
1-way (m)	0.61%	9.63%	3.76%	9.26%	0.19%	4.9%
2-way (m)	0.21%	9.10%	3.32%	9.18%	0.03%	4.8%
4-way (m)	0.21%	9.10%	3.31%	9.15%	0%	4.8%
go						
# of Acc	1548650	539976	2088626	1441866	790450	2232316
1-way (m)	2.04%	2.95%	2.28%	6.32%	1.12%	3.29%
2-way (m)	1.52%	2.41%	1.75%	5.71%	0.32%	2.52%
4-way (m)	1.46%	2.33%	1.69%	4.69%	0.04%	1.94%

Table 7.10: L1 D-cache performance in DLX and PERL

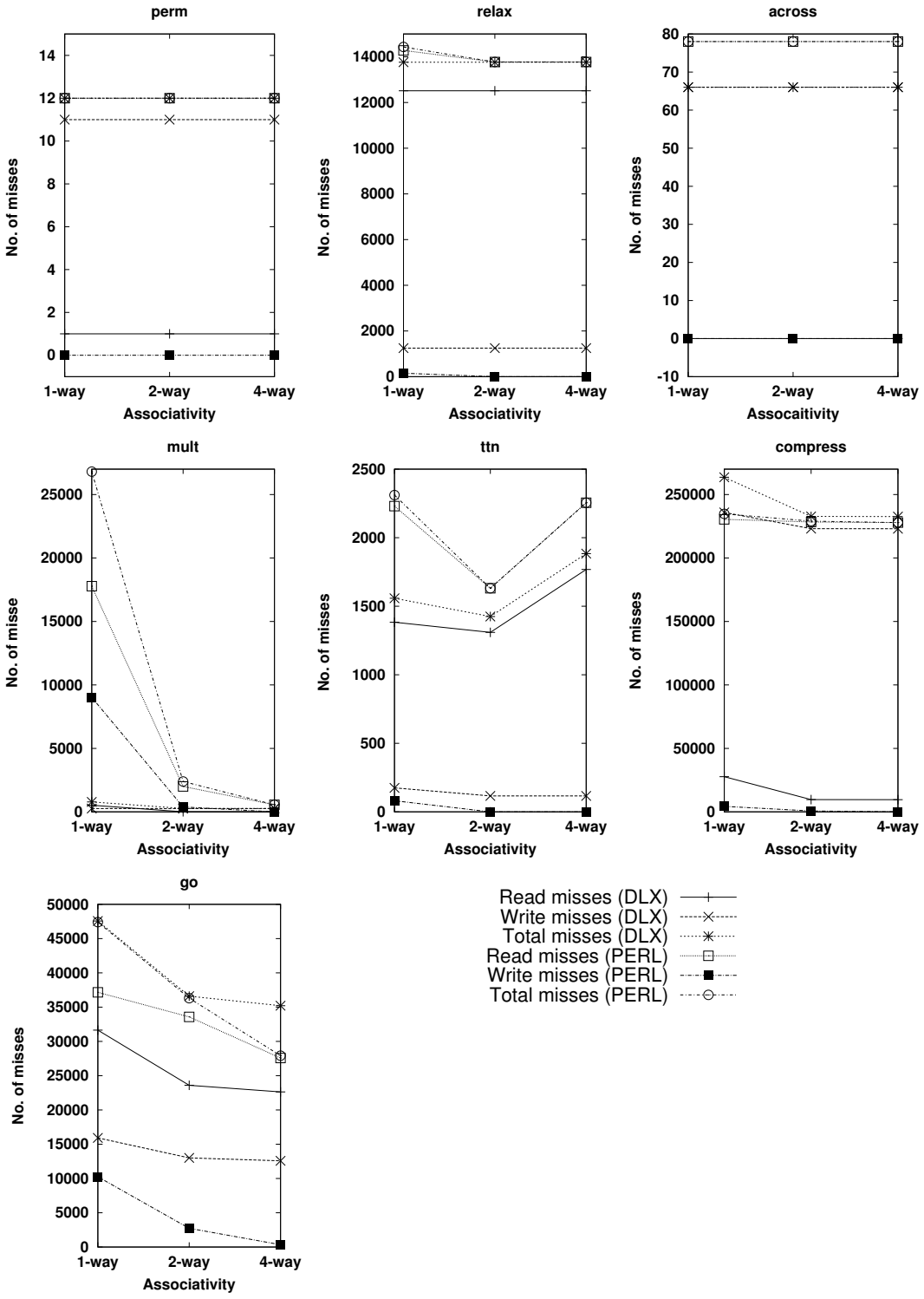


Figure 7.5: Number of L1 D-cache misses with varying associativity

Program/ Machine	Metric	1-way		2-way		4-way	
		# Acc	# miss	#Acc	#miss	#Acc	#miss
relax DLX	Reads	13764	1157	13764	1166	13764	1247
	Writes	13261	227	13261	118	13261	12
	Ifetch	24	24	24	24	24	24
	Total	27049	1408	27049	1308	27049	1283
relax PERL	Reads	14430	1199	13772	1182	13772	1249
	Writes	13554	127	13269	87	13269	44
	Ifetch	27	27	27	27	27	27
	Total	28011	1353	27068	1296	27068	1320
ttn DLX	Reads	1558	199	1425	199	1884	198
	Writes	305	50	141	32	137	40
	Ifetch	2070	637	2070	637	2070	637
	Total	3933	886	3636	868	4091	875
ttn PERL	Reads	2311	199	1631	200	2255	200
	Writes	491	46	134	23	136	37
	Ifetch	11033	1151	11033	1152	11033	1152
	Total	13835	1396	12798	1375	13424	1389
compress DLX	Reads	263589	99572	232733	98978	232627	98474
	Writes	245010	6379	229630	4736	229561	3188
	Ifetch	1566	1467	1566	1443	1566	1467
	Total	510165	107418	463929	105157	463754	103129
compress PERL	Reads	234773	107033	229050	104620	227892	103987
	Writes	229320	3602	226174	2826	225284	1571
	Ifetch	29908	6460	29908	6445	29908	6460
	Total	494001	117095	485132	113891	483084	112018
go DLX	Reads	47564	14682	36603	14470	35207	14457
	Writes	19678	2271	15348	2533	14838	2247
	Ifetch	27681	4843	27681	4965	27681	4882
	Total	94923	21796	79632	21968	77726	21586
go PERL	Reads	47406	15303	360305	14960	27916	14123
	Writes	19321	1993	18443	1729	17924	1677
	Ifetch	38989	16424	38989	15414	38989	14842
	Total	105716	33720	417737	32103	84829	30642

Table 7.11: L2 cache misses for varying associativity in L1 D-cache

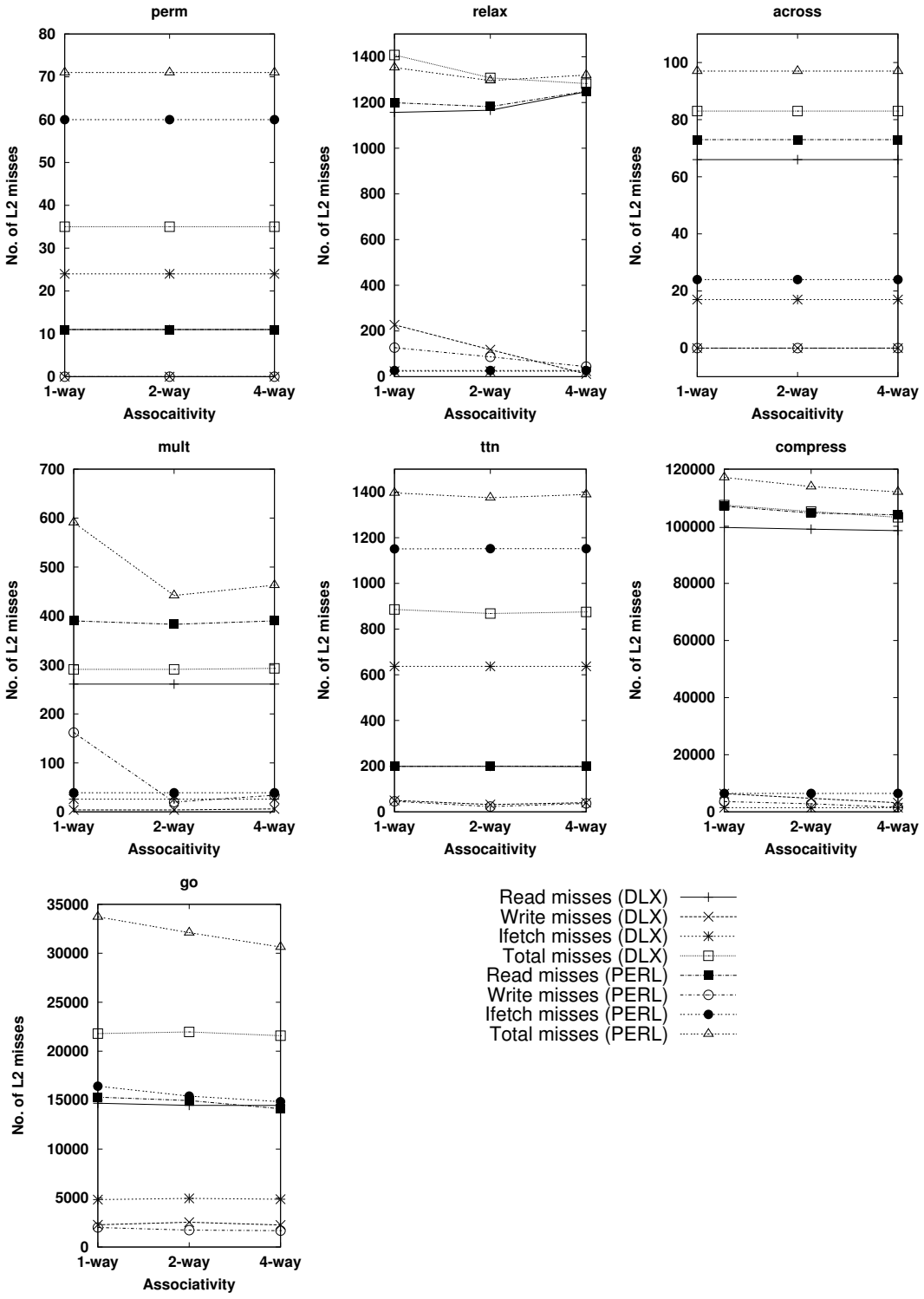


Figure 7.6: Number of L2 misses with varying associativity in L1 D-cache

(except for *relax* benchmark). This means that PERL in general requires fewer cycles to serve its misses than the DLX.

Program	No. of requests served	
	PERL	DLX
perm	3004	1453
relax	1	1478
across	201	5
mult	108813	11
ttn	785443	77834
compress	2484791	276313
go	799432	487988

Table 7.12: Number of requests served by Load-all-wide technique

7.3.6 Effect of Misses and Bank Clashes

The impact of misses is measured by finding the number of extra cycles required to process them (table 7.13 and figure 7.7). Similarly the number of cycles required to serve the bank clashes are presented in table 7.14 and figure 7.7. The number of bank clashes in PERL was found to be zero for all benchmarks.

PERL requires fewer cycles to process its misses than the DLX for *across*, *relax* and *compress* benchmark programs. However DLX requires fewer cycles to serve misses in *perm*, *mult* and *ttn* benchmarks.

7.3.7 Impact on Execution Time

The execution time presented in the previous chapter assumed a perfect memory for both PERL and DLX. The number of extra cycles required to process the misses and bank clashes were not accounted for. However, they should be added to the execution time in order to get the execution time including the time to access memory.

We present the modified execution time required in table 7.15. The execution time shown here are for *md4* model with *bp* and *bs* branch prediction schemes in

program	machine	1-way	2-way	4-way
		# of cycles	# of cycles	# of cycles
perm	DLX	463	463	463
	PERL	832	832	832
relax	DLX	59715	58734	57780
	PERL	9288	16928	17027
across	DLX	1079	1079	1079
	PERL	429	429	429
mult	DLX	5802	3738	3756
	PERL	56461	5209	1743
ttn	DLX	22180	21558	23493
	PERL	60056	59120	60418
compress	DLX	1158225	1014425	1002679
	PERL	228543	212546	206195
go	DLX	406676	367845	359219
	PERL	811484	723998	719440

Table 7.13: Extra cycles required to serve misses

Program	# of cycles
perm	30
relax	980
across	48
mult	4096
ttn	20630
compress	29737
go	85788

Table 7.14: Extra cycles required to serve bank clashes in DLX

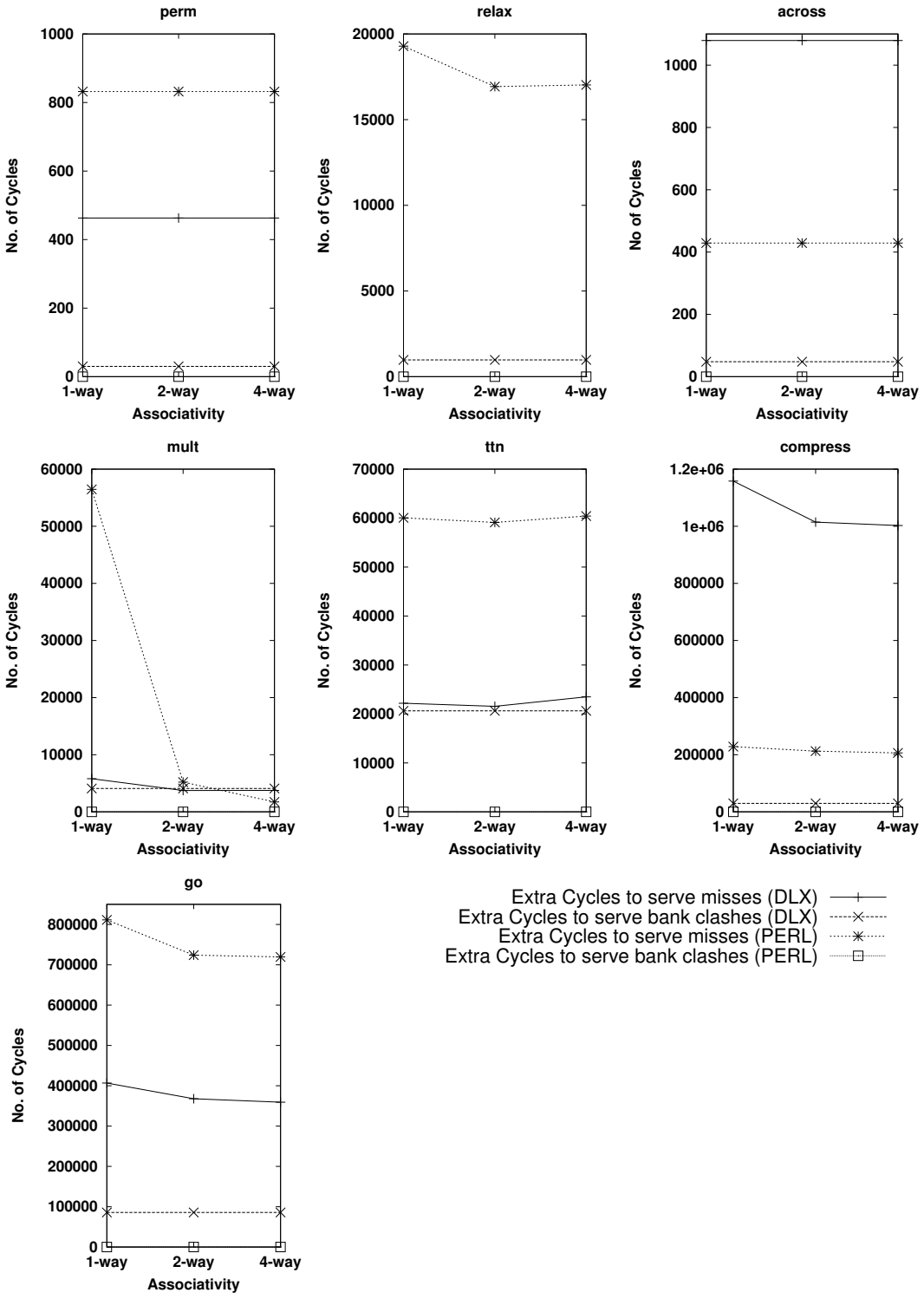


Figure 7.7: Extra cycles required to service misses and bank clashes

Program	Mach.	Ex.Time	Extra cycles		Corr Ex.Time	Speed Up
			BC	Miss		
perm	DLX	6740	463	30	7233	1.027
	PERL	**6210**	832	0	**7042**	
relax	DLX	604554	57780	980	663314	1.125
	PERL	**572601**	17027	0	**589628**	
across	DLX	**2336**	1079	48	**3463**	0.980
	PERL	3105	429	0	3534	
mult	DLX	**240182**	3756	4096	248034	1.004
	PERL	245272	1743	0	**247015**	
ttn	DLX	3680393	23493	20630	3724516	1.224
	PERL	**2982786**	60418	0	**3043204**	
compress	DLX	6838387	1002679	29737	7870803	1.227
	PERL	**6209845**	206195	0	**6416040**	
go	DLX	3636048	359219	85788	4081055	
	PERL	**2593769**	719440	0	**3313209**	1.231

Table 7.15: Execution time including the time to access memory

DLX and PERL respectively. The extra cycles required is taken with 4-way set associative L1 data cache.

After compensating for the cycles required to process misses and bank clashes PERL performs better than DLX in all programs except *across*. A notable observation is that PERL performs better than DLX for *mult* program for which DLX was earlier performing better than PERL.

After the detailed execution simulation followed by the cache memory simulation, we find PERL performs better than DLX with most of the programs. As some programs considered were small the speed up obtained is not pronounced. The two programs *compress* and *ttn* which were reasonably large programs have shown better speed up than other programs.

Chapter 8

Conclusions

In this thesis, we presented a case for Performance Enhanced Register–Less (PERL) RISC processor. With improvement in memory subsystem architecture, the access time of on-chip caches have come very close to that of the registers. The performance obtained due to bigger on-chip caches is very impressive, whereas the same cannot be said of registers. Having more registers has not shown any significant improvement in the performance.

In the current situation, we argue that the time has come to re-look and redesign the instruction set architecture. A pure memory-to-memory instruction set is an ideal choice as it gives the maximum code compaction. Compilers can do away with complex register allocation process and instead work on techniques to extract more ILP.

8.1 Contributions

In this thesis, we surveyed the benefits of cache memory and registers. A new technique called "On-Chip registering of memory location" is proposed. The concept has been shown to be of great help in implementing a pure memory-to-memory processor. We use this technique as the first level of cache in PERL (L0 cache).

An analytical model of PERL was presented considering the worst case memory requirement of PERL. We compared the CPI of DLX and PERL using this model

and show that at higher hit ratios PERL would outperform DLX.

A hypothetical memory-to-memory instruction set for PERL was presented along with a hypothetical superscalar pipeline to implement the proposed instruction set. A new branch prediction scheme was proposed to predict indirect branches using a pair of stacks. This technique is expected to predict the call/return instructions with an accuracy close to 100%.

We used the instruction set simulators to execute seven benchmark programs chosen from SPEC95 and NASA NAS test suites. We assumed a perfect cache during the execution of programs on simulators and collected the memory traces. The traces were then analyzed off-line to obtain the performance of cache memory subsystem.

The simulations show that PERL requires fewer cycles than DLX to execute most benchmark programs. The simulations also show that PERL consistently executes fewer instructions (6% to 65%) than the DLX. The indirect branch prediction scheme used succeeded in predicting with a success rate of close to 100%.

The off-line memory trace analysis shows that the average number of memory accesses per instruction in a superscalar PERL (order 4) is about 2–2.5. The L0 cache was able to provide 21%-97% hit ratio. For some programs, it was found that the number of memory references that actually go to L1 cache is smaller in PERL than in DLX. The extra cycles required to process the misses and bank clashes were more in DLX than in PERL for most of the programs. Further, because of L0 cache PERL does not have any bank clashes. The load-all-wide technique is found to benefit PERL more than the DLX. Large instruction cache block size and large instruction cache is also found to benefit PERL more than the DLX.

The effective execution time after considering the performance of cache sub system was determined. The results show clearly that PERL outperforms DLX in all but one program. The main contribution for PERL to perform better is certainly the instruction count, which in turn is a direct implication of the memory-to-memory instruction set and the compiler.

8.2 Future Work

Each instruction in PERL, as proposed, is very long (128 bits). This results in a bigger static code size of programs, which on an average is twice the code size of programs in a register-to-register machine. This puts enormous pressure on instruction cache bandwidth. We feel that some mechanisms for code compaction are possible and should be investigated.

Some resources in PERL such as L0 cache, reorder buffer etc., are complex and their effect on clock cycle time has to be investigated. Finding optimal sizes of these resources is also an interesting work.

The *gcc* compiler is perhaps not the best compiler for memory-to-memory instruction set. Building a compiler for PERL would be more beneficial. While the machine specific optimizations for register architectures are well known and implemented by the compilers, the same was not done for PERL. Machine specific optimizations provide good performance enhancements. Some machine dependent optimizations for PERL should also be identified and implemented.

8.2.1 Some Possible Optimizations

The results clearly show that a large percentage of cycles in PERL are fetch and decode stalls. Further, it is also observed that most of the fetch stalls are due to filled up instruction queue. This indicates that the decode unit is not processing instructions at the same rate as the fetch unit. This is primarily because of data dependencies. Techniques like loop unrolling can be adopted efficiently in PERL to extract more ILP as there is no register pressure. Techniques like instruction reordering will definitely result in reduced number of fetch and decode stalls resulting in more ILP.

Apart from these, cache prefetching techniques can also be used to reduce the impact of cache misses.

Based on this study and results for the benchmark programs, we conclude that a memory-to-memory architecture like PERL is definitely an attractive and viable method to obtain better performance than register-to-register architecture using the current technology.

Bibliography

[1] David A. Patterson and David R. Ditzel, "The Case for the Reduced Instruction Set Computer," *Computer Architecture News, ACM SIGARCH*, vol. 8, no. 6, pp. 25–33, Oct 1980.

[2] Dell Computers, *IA-64 Architecture and ITANIUM^{TM}*, White paper, April 1999.

[3] Sun Microsystems, *UltraSPARC III Cu User's manual*, May 2002.

[4] Tim Fu, Farshid Iravani, Mahdi Seddighnezhad, Kenneth Yeager, and David Zhang, "R18000 The latest SGI^{TM} Superscalar Microprocessor," *Hot Chips*, Aug 2001.

[5] Hewlett Packard, *PA-RISC 8X00 Family of microprocessors with Focus on PA-8700*, Technical white paper, April 2000.

[6] David A. Patterson and R. Piepho, "RISC assessment. A high-level language experiment," *Proceedings of the 9th annual Symposium on Computer Architecture, ACM SIGARCH*, pp. 3–8, Apr 1982.

[7] David A. Patterson, "Reduced Instruction Set Computers," *Communications of the ACM*, vol. 28, no. 1, pp. 8–21, Jan 1985.

[8] Mark Brehob, Travis Doom, Richard Enbody, William H. Moore, Sherry Q. Moore, Ron Sass, and Charles Severance, "Beyond RISC - The Post-RISC Architecture," Tech. Rep. CPS-96-11, Michigan State University, Department of Computer Science, Mar 1996.

[9] Dileep Bhandarkar and Douglas W. Clark, "Performance from Architecture: Comparing a RISC and a CISC with Similar Hardware Organization," *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 310–319, Apr 1991.

[10] David A. Patterson and Carlo H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer," *Proceedings of the 8th annual Symposium on Computer Architecture*, pp. 443–457, May 1981.

[11] Wulf W, "Compilers and Computer Architecture," *IEEE Computer*, vol. 14, no. 7, pp. 41–47, July 1981.

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

[13] Dave Christie, "Developing The AMD-K5 Architecture," *IEEE Micro*, vol. 16, no. 2, pp. 16–26, Apr 1996.

[14] MOTOROLA, IBM and APPLE, *Power PC 601 RISC Microprocessor User's Manual*, 1993.

[15] Tim Horel and Gary Lauterbach, "UltraSPARC-III:Designing Third-Generation 64-Bit Performance," *IEEE Micro*, vol. 19, no. 3, pp. 73–85, 1999.

[16] David B. Papworth, "Tuning the Pentium Pro Microarchitecture," *IEEE Micro*, vol. 16, no. 2, pp. 8–15, Apr 1996.

[17] Intel Literature center, *IA-32 Intel Architecture Software Developer's Manual Manual Volume 1: Basic Architecture*, 2001.

[18] Advanced Micro Devices Inc, *AMD ATHLON Processor Technical Brief*, Dec 1999.

[19] Linley Gwennap, "Alpha 21364 to Ease Memory Bottlenec," *Cahners MicroDesign Resources, Microprocessor Report*, Oct 1998.

[20] B. Ramakrishna Rau and Joseph A. Fisher, "Instruclion-Level Parallel Processing: History, Overview, and Perspective," *The Journal of Supercomputing*, vol. 7, no. 1/2, pp. 9–50, 1993.

[21] Mike Johnson, *Superscalar Microprocessor Design*, Prentice Hall Series in Innovative Technology, PTR Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[22] John L. Hennessy and David A.Patterson, *Computer Architecture A Quantitative Approach*, Morgan Kaufmann Publishers, INC, San Mateo, California, 1991.

[23] Kogge P.M., *The Architecture of Pipelined computers*, McGraw-Hill, New York, 1981.

[24] D.W. Anderson, F.J. Sparacio, and R.M. Tomasulo, "The IBM/360 Model 91: Machine Philosophy and Instruction-Handling," *IBM Journal of Research and Development*, vol. 11, pp. 8–24, Jan 1967.

[25] R.M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, vol. 11, pp. 25–33, Jan 1967.

[26] John H.E. Edmondson, Paul Rubinfeld, and Ronald Preston, "Superscalar Instruction Execution in the 21164 Alpha Microprocessor," *IEEE Micro*, vol. 15, no. 2, pp. 33–43, Apr 1995.

[27] IBM Microelectronics, Motorola, *Advance Information PowerPC* 620^{TM} *RISC Microprocessor Technical Summary*, 1994.

[28] Norman P. Jouppi and David W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machine," Tech. Rep. WRL-TR-89.7, Digital Western Research Laboratory, July 1989.

[29] Daniel Tabak, *Advanced Microprocessors*, McGraw-Hill, New York, second edition, 1991.

[30] Fisher J. A., "Very Long Instruction Word Architectures and the ELI-512," *Conference Proceedings of the 10th annual International Symposium on Computer Architecture (SIGARCH'83), ACM SIGARCH*, pp. 140–150, Jun 1983.

[31] M. S. Schlansker and B. R. Rau, "EPIC: Explicitly Parallel Instruction Comupting," *IEEE Computer*, vol. 33, no. 2, pp. 37–45, Feb 2000.

[32] Wm. A. Wulf and Sally A. Mckee, "Hitting the Memory Wall: Implications of the Obvious," *ACM SIGARCH, Computer Architecture News*, vol. 23, no. 1, pp. 20–24, Mar 1995.

[33] Maurice V. Wilkes, "The Memory Wall and the CMOS End - Point," *ACM SIGARCH, Computer Architecture News*, vol. 23, no. 4, pp. 4–6, Sep 1995.

[34] Michael Butler, Tse-Yu Yeh, Yale Patt, Mitch Alsup, Hunter Scales, and Michael Shebanow, "Single Instruction Stream Parallelism Is Greater than Two," *Proceedings of the 18th annual International Symposium on Computer Architecture (ISCA'91), ACM SIGARCH*, pp. 276– 286, May 1991.

[35] David W. Wall, "Limits of Instruction Parallelism," *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 176–188, Apr 1991.

[36] Michael D. Smith, Mike Johnson, and Mark A. Horowitz, "Limits on Multiple Instruction Issue," *Proceedings of 3rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pp. 290–302, Apr 1989.

[37] Matthew A. Postiff, David A. Greene, Gary S. Tyson, and Trevor Mudge, "The Limits of Instruction Level Parallelism in SPEC95 Applications," *Computer Architecture News, ACM SIGARCH*, vol. 27, no. 1, pp. 31–34, Mar 1999.

[38] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[39] Lam M., "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," *In Proceedings., ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pp. 318–327, Jun 1987.

[40] Backus J., "Can Programming Be Liberated From the Von Neumann Style? A Functional Style and Its Algebra of Programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, Aug 1978.

[41] Teresa Monreal, Antonio Gonzalez, Mateo Valero, Jose Gonzalez, and Victor Vinals, "Dynamic Register Renaming Through Virtual-Physical Registers," *MICRO 32, Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 186–192, Nov 1999.

[42] John A. Swenson and Yale N. Patt, "Hierarchial Registers for Scientific Computers," *In Proceedings of the International Conference on Supercomputing*, pp. 346–343, Jul 1988.

[43] Robert Yung and Neil C. Wilheim, "Caching Processor General Registers," *International Conference on Computer Design*, pp. 307–312, Oct 1995.

[44] Robert Yung and Neil C. Wilheim, "Caching Processor General Registers," *Sun Microsystems Laboratories Technical Report*, Jun 1995.

[45] Matthew A. Postiff, David A. Greene, Steven Raash, and Trevor Mudge, "Integrating Superscalar Processor Components to Implement Register Caching," *Proc. 15th ACM International Conference on Supercomputing (ICS'01), ACM SIGARCH*, pp. 348–357, 2001.

[46] Soo-Mook Moon and Kemal Ebcioglu, "A Study of the Number of Memory Ports in Multiple Instruction Issue Machines," *MICRO 26, Proceedings of the 26th Annual International Symposium on Microarchitecture*, pp. 49–58, Dec 1993.

[47] Jan Hoogerbrugge and Henk Corporaal, "Register File Port Requirements of Transport Triggered Architecture," *MICRO 27, Proceedings of the 27th*

Annual ACM/IEEE International Symposium on Microarchitecture, pp. 191–195, Nov 1994.

[48] Dezso Sima, "The Design Space of Register Renaming Techniques," *IEEE Micro*, vol. 20, no. 5, pp. 70–83, Sep/Oct 2000.

[49] Todd M. Austin and Gurinder S. Sohi, "Dynamic Dependency Analysis of Ordinary Programs," *Proceedings of the 19th annual International Symposium on Computer Architecture (ISCA'92), ACM SIGARCH*, May 1992.

[50] G.D. McNiven and E.S. Davidson, "Analysis of Memory Reference behavious For Design of Local Memories," *Proceedings of the 15th annual International Symposium on Computer Architecture (ISCA'88), ACM SIGARCH*, pp. 56–63, 1988.

[51] Andrew S. Huang and John Paul Shen, "The Intrinsic Bandwidth Requirements of Ordinary Programs," *Proceedings of 7nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pp. 105–114, Oct 1996.

[52] Keith I. Farkas, Paul Chow, Norman P. Jouppi, and Zvonko Vranesic, "Memory-System Design Considerations for Dynamically-Scheduled Processors," *Proceedings of the 24th annual International Symposium on Computer Architecture (ISCA'97), ACM SIGARCH*, pp. 133–143, Jul 1997.

[53] Richard L. Sites, "How to use 1000 Registers," *Proceeding of 1st Caltech Conference on VLSI*, pp. 527–532, 1979.

[54] Jack W. Davidson and Richard A. Vaughan, "The Effect of Instruction Set Complexity on Program Size and Memory Performance," *Proceedings of 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, pp. 60–64, Oct 1987.

[55] Norman P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," *Proceedings of the 17th annual International Symposium on Computer Architecture (ISCA'90), ACM SIGARCH*, pp. 364–373, Jun 1990.

[56] Gurindar S. Sohi and Manoj Franklin, "High-Bandwidth Data Memory Systems for Superscalar Processors," *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 53–62, Apr 1991.

[57] Andrew Wolfe and Rodney Boleyn, "Two-ported Alternatives for Superscalar Processors," *MICRO 26, Proceedings of the 26th Annual International Symposium on Microarchitecture*, pp. 41–48, Dec 1993.

[58] Kenneth M. Wilson, Kunle Olukotun, and Mendel Rosenblum, "Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessor," *Proceedings of the 23rd annual International Symposium on Computer Architecture (ISCA'96), ACM SIGARCH*, pp. 147–157, May 1996.

[59] Kenneth M. Wilson and Kunle Olukotun, "Designing High Bandwidth On-Chip Caches," *Proceedings of the 24th annual International Symposium on Computer Architecture (ISCA'97), ACM SIGARCH*, pp. 121–132, Jul 1997.

[60] Lizy Kurian John, Raghuveer Reddy, Vijay Kammila, and Peter Maurer, "Investigating the Use of Cache as a local Memory," *1st IEEE Symposium on High–Performance Computer Architecture*, pp. 117–122, Dec 1995.

[61] Philip Machanick, "The Case for SRAM Main Memory," *Computer Architecture News, ACM SIGARCH*, vol. 24, no. 5, pp. 23–30, Dec 1996.

[62] Michel Cekleov and Michel Dublois, "Virtual-Address caches," *IEEE Micro*, vol. 17, no. 5, pp. 64–71, Sep/Oct 1997.

[63] D. Burger and J. Goodman, "Billion-Transistor Architecures," *IEEE Computer*, vol. 30, no. 9, pp. 46–57, Sep 1997.

[64] Ashley Saulsbury, Fong Pong, and Andreas Nowatzyk, "Missing the Memory Wall: The Case for Processor/Memory Integration," *Proceedings of the 23rd annual International Symposium on Computer Architecture (ISCA'96), ACM SIGARCH*, pp. 90–101, May 1996.

[65] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick, "A Case For Intelligent RAM: IRAM," *IEEE Micro*, vol. 17, no. 2, pp. 34–44, Apr 1997.

[66] Doug Burger, James R. Goodman, and Alain Kagi, "Limited Bandwidth To Affect Processor Design," *IEEE Micro*, pp. 55–62, Nov/Dec 1997.

[67] Freedom CPU Project, Draft and Request for comment, Patch YG 2001.1.1.14, *FCPU MANUAL REV.0.2.2.* , 2001.

[68] Sun Microsystems, *An Overview of UltraSPARC III Cu Processor*, A white paper, Jun 2002.

[69] P. Bannon, ""alpha 21364 – a scalable single–chip smp"," *Intel Microprocessor Forum*, Oct 1998.

[70] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel, "The Microarchitecture of the **Pentium**^{R} 4 Processor," *Intel Technology Journal Q1*, 2001.

[71] Intel Corporation, *IA-64 Application Developer's Architecture guide*, May 1999.

[72] Harsh Sharangpani, ""intel itanium^{TM} processor micorarchitecture overview"," *Intel Microprocessor Forum*, Oct 1999.

[73] Alexander Klaiber, "The Technology Behind CRUSOE^{TM} Processors," Tech. Rep., Transmeta Corporation, Jan 2000.

[74] Vinod Cuppu, Bruce Jacob, Brian Davis, and Trevor Mudge, "A Performance Comparison of Contemporary DRAM Architectures," *Proceedings of the 26th*

annual International Symposium on Computer Architecture (ISCA'99), ACM SIGARCH, pp. 222–233, May 1999.

[75] Brian Davis, Bruce Jacob, and Trevor Mudge, "The New DRAM Interfaces: SDRAM, DRDRAM and Variants," *In High Performance Computing, Lecture Notes in Computer Science*, pp. 25–31, Oct 2000.

[76] Alan J. Smith, "Cache Memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, Sep 1982.

[77] Steven Przybylski, Mark Horowitz, and John L. Hennessy, "Performance Tradeoffs in Cache design," *Proceedings of the 15th annual International Symposium on Computer Architecture (ISCA'88), ACM SIGARCH*, pp. 290–298, May 1988.

[78] Mark D. Hill and Alan Jay Smith, "Experimental Evaluation of On-Chip Microprocessor Cache Memories," *Proceedings of the 11th annual International Symposium on Computer Architecture (SIGARCH)*, pp. 158–166, Jun 1984.

[79] Richard J.Eickmeyer and Janak H.Patel, "Performance Evaluation of On-Chip Register and Cache Organizations," *Proceedings of the 15th annual International Symposium on Computer Architecture (ISCA'88), ACM SIGARCH*, pp. 64–70, 1988.

[80] Linley Gwennap, "Digital 21264 Sets New Standard," *Cahners MicroDesign Resources, Microprocessor Report*, vol. 10, no. 14, Oct 1996.

[81] James R. Goodman and Wei Chung Hsu, "On the Use of Registers Vs. Cache to Minimize Memory Traffic," *Proceedings of the 13th annual International Symposium on Computer Architecture (ISCA'86), ACM SIGARCH*, pp. 375–383, Jun 1986.

[82] Hwu W and P. P. Chang, "Exploiting parallel Microprocessor Microarchitecture With a Compiler Code Generator," *Proceedings of the 15th annual International Symposium on Computer Architecture (ISCA'88), ACM SIGARCH*, pp. 45–53, Jun 1988.

[83] David G. Bradlee, Susan J. Eggers, and Robert R. Henry, "The Effect on RISC Performance of Register Set Size and Structure Versus Code Generation Strategy," *Proceedings of the 18th annual International Symposium on Computer Architecture (ISCA'91), ACM SIGARCH*, pp. 330–339, May 1991.

[84] Tokuzo Kiyohara and John C Gyllenhaal, "Code Scheduling for VLIW/Superscalar Processors With Limited Register Files," *MICRO 25, Proceedings of the 25th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture*, pp. 197–201, Dec 1992.

[85] David G. Bradlee, Susan J. Eggers, and Robert R. Henry, "Integrating Register Allocation and Instruction Scheduling for RISCs," *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 122–131, Apr 1991.

[86] Manoj Franklin and Gurindar S. Sohi, "Register traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors," *MICRO 25, Proceedings of the 25th Annual ACM/IEEE International Symposium and Workshop on Microarchitecture*, pp. 236–245, Dec 1992.

[87] Kishor S. Trivedi, *Probability & Statistics with Reliability, Queuing and Computer Science Applications*, Prentice Hall, Inc, Englewood Cliffs, N.J., U.S.A, 1982.

[88] D. Alpert, A. Averbuch, and O. Danieli, "Performance Comparison of Load/Store and Symmetric Instruction Set Architectures," *Proceedings of the 17th annual International Symposium on Computer Architecture (ISCA'90), ACM SIGARCH*, pp. 172–181, Jun 1990.

[89] Amund Lunde, "Empirical Evaluation of Some Features of Instruction Set Processor Architecture," *Communications of the ACM*, vol. 20, no. 3, pp. 143–153, Mar 1977.

[90] Douglas W. Clark and Henry M. Levy, "Measurement and Analysis of Instruction Set Use in the VAX-11/780," *Proceedings of the 9th annual Symposium on Computer Architecture, ACM SIGARCH*, pp. 9–17, Apr 1982.

[91] Cheryl A. Wiecek, "A Case Study of VAX-11 Instruction Set Usage for Compiler Execution," *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-I)*, pp. 177–184, Mar 1982.

[92] Robert Cmelik, Shing I. Kong, David R. Ditzel, and Edmund J. Kelly, "An Analysis of MIPS and SPARC Instruction Set Utilization on the SPEC Benchmarks," *Proceedings of 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pp. 290–302, Apr 1991.

[93] Po-Yung Chang, Eric Hao, and Yale N. Patt, "Target Prediction for Indirect Jumps," *Proceedings of the 24th annual International Symposium on Computer Architecture (ISCA'97), ACM SIGARCH*, pp. 274–283, Jul 1997.

[94] P. Suresh and Rajat Moona, "PERL - A Registerless Architecture," *Processdings of The 5th International Conference of High Performance Computing*, Dec 1998.

[95] J.K.F. Lee and A.J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer*, vol. 17, no. 1, pp. 6–22, Jan 1984.

[96] David R. Kaeli and Philip G. Emma, "Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns," *Proceedings of the 18th annual International Symposium on Computer Architecture (ISCA'91), ACM SIGARCH*, pp. 34–42, May 1991.

[97] Cecile Moura, "SuperDLX–A Generic Superscalar Simulator," Masters thesis, Advanced compilers, Architecture and Parallel Systems Group, McGill University, May 1993.

[98] T. S. Balaji, "Design and Simulation of PERL RISC," Masters thesis, Indian Institute of Technology Kanpur, Kanpur, Feb 1997.

[99] G. V. Ramana Kumar, "C compiler and cache performance studies for perl risc processor," Masters thesis, Indian Institute of Technology Kanpur, Kanpur, Feb 1997.

[100] Richard M. Stallman, *Using and Porting GNU C Compiler*, Free Software Foundation, 1992.

[101] New Mexico State University, *ACS: Cache Simulator*.

[102] http://www.specbench.org, *SPEC95 CPU benchmark*.

[103] David H. Bailey and John T. Barton, *THE NAS Kernel Benchmark Program, Numerical*, Aerodynamic Simultions Systems Division, NASA, Ames Research center, Aug 1986.

[104] Samson Belayneh and David R. Kaeli, "A Discussion on Non-Blocking/Lockup-Free Caches," *Computer Architecture News, ACM SIGARCH*, vol. 24, no. 3, pp. 18–25, Jun 1996.

[105] Michael Butler and Yale Patt, "The Effect of Real Data Cache Behavior on the Performance of a Microarchitecture That Supports Dynamic Scheduling," *MICRO 24, Proceedings of the 24th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 34–41, Nov 1991.

Appendix A

PERL Instruction Set

In this appendix, we present the instruction architecture of PERL. The PERL instruction set is designed primarily for the studies in this thesis and no emphasis is given on the code density reduction etc.

A.1 Instruction Format

PERL instruction set is based on memory-to-memory architecture. A three address format is used for all instructions. The general format of PERL instructions bit encoding is as shown in the figure A.1.

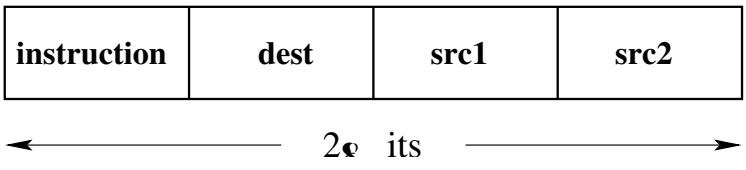


Figure A.1: PERL instruction format

A.1.1 Non-Branch Instructions

All PERL non-branch instructions have three operands. The format of 3 operand instructions is as shown below.

```
instruction_mnemonic dest<:dType>, src1<:dType>, src2<:dType>
```

In this format, the instruction_mnemonic can refer to any of the instructions given in table A.1. The dType field in the instruction depends upon the kind of the instruction. There are eight integer data types and two floating point data types as shown in table A.2. In the first example below the data types of dest, src1 and src2 are specified individually. Whereas, a shorter form is used in second example where src1, src2 and dest have the same dTypes.

```
1.  add  _p:ub4, _q:b2, _r:b1 ;;data types specified individually
2.  addb4 _p, _q, _r          ;;same data type for dest,src1 and src2
```

The dest, src1 and src2 specify the destination and source operands each in any one of the 4 addressing modes as specified in table A.3. In the first example below dest, src1 and src2 are specified using direct, base and indirect addressing modes respectively. Whereas in the second example the dest, src1 and src2 are specified using base, direct and immediate addressing modes respectively. In the third example the dest is specified by direct address and is a 4 byte integer, while src1 and src2 are both 1 byte integers specified by base and indirect addressing modes respectively.

```
1.   addb4  _p, 28(SP), @_square
2.   mulb4  8(SP), _q, #100
3.   add    _p:b4, 32(SP):b1, @_comm:b1
```

If an immediate addressing mode is used to specify the destination, then the result is not saved and this instruction then behaves as NOP. However, the source operands are fetched by the processor and the operation is performed. Such instructions could be used for prefetching the operands into cache by the compilers.

A.1.2 Branch Instructions

PERL processor supports unconditional and conditional branch instructions, each with direct or indirect addressing mode. There is also a trap instruction to support calls for the operating system.

The format of unconditional jumps is as follows.

```
j target, savePC
```

Here target field specifies the address to which control has to be transferred, savePC is an optional field. PERL implements procedure calls using jump instructions in which the current PC value is saved in location savePC. In the instruction's binary coding target is stored in dest field whereas savePC is stored in src1. The field src2 of instruction coding is not used.

The conditional jump instructions use 3 operands. The target is specified in the dest field. Other two operands are used to evaluate the condition. If the condition evaluates to true, control is transferred to the target.

The following is an example of a conditional jump. In this instruction, if 4 byte signed number `_total` is less than 4 byte constant `100`, the control is transfered to location `_target`.

```
jltb4 _target, _total, #100
```

Trap instruction has only one operand that provides the trap number. This operand is stored in the dest field of the instruction. Traps are typically used for library and operating system calls. An example below shows how a call to an operating system function `_exit` is implemented in PERL.

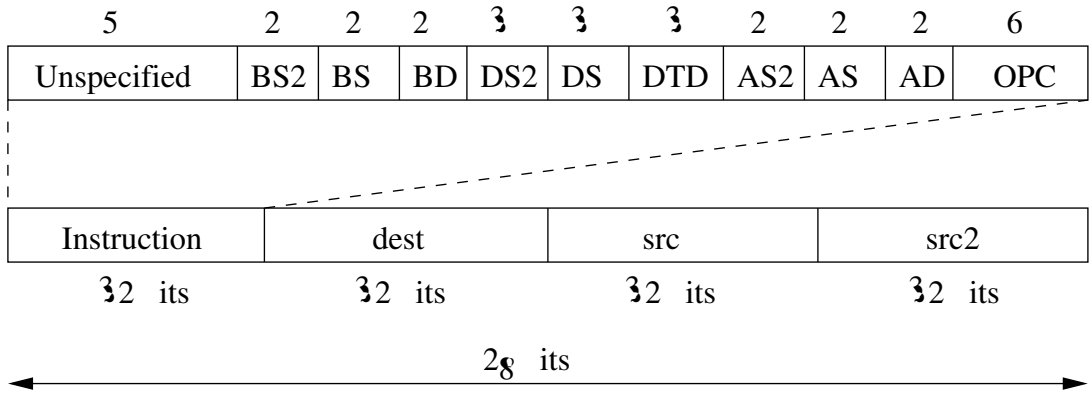
```
        j _exit
_exit:
        trap #0
        j -8(sp)
```

A.2 Instruction Encoding

Figure A.2 shows the encoding scheme of PERL instruction.

A.2.1 Opcodes

PERL instructions are broadly classified into Integer and Floating point class. Table A.1 gives a 6–bit encoding scheme for all the instructions in PERL. This scheme facilitates an early detection of Jump and Branch instructions.



OPC	Operation Code
AD, AS & AS2	Addressing odes for destination and source operands
DTD, DS & DS2	Data types of destination and source operands
BD, BS & BS2	Base pointers for destination and source operands

Figure A.2: PERL instruction encoding

CLASS	**6–bit encoding**						Value	**Mnemonic**	**Function Class**
Integer	0	0	0	0	0	0	0	NOP	NONE
Arithmetic &	0	0	0	0	0	1	1	ADD	INT_ALU
Logical	0	0	0	0	1	0	2	SUB	INT_ALU
	0	0	0	0	1	1	3	SLL	INT_SHIFT
	0	0	0	1	0	0	4	SRL	INT_SHIFT
	0	0	0	1	0	1	5	SRA	INT_SHIFT
	0	0	0	1	1	0	6	OR	INT_ALU
	0	0	0	1	1	1	7	AND	INT_ALU
	0	0	1	0	0	0	8	XOR	INT_ALU
	0	0	1	0	0	1	9	MUL	INT_MUL
	0	0	1	0	1	0	10	DIV	INT_DIV

...continued

CLASS	6–bit encoding						Value	Mnemonic	Function Class
Control	0	1	0	0	0	0	16	J	INT_BRANCH
	0	1	0	0	0	1	17	JEQ	INT_BRANCH
	0	1	0	0	1	0	18	JNE	INT_BRANCH
	0	1	0	0	1	1	19	JGT	INT_BRANCH
	0	1	0	1	0	0	20	JLT	INT_BRANCH
	0	1	0	1	0	1	21	JGE	INT_BRANCH
	0	1	0	1	1	0	22	JLE	INT_BRANCH
Floating Point Arithmetic	1	0	0	0	0	1	33	ADDF	FP_ADD
	1	0	0	0	1	0	34	SUBF	FP_ADD
	1	0	1	0	0	1	35	MULF	FP_MUL
	1	0	1	0	1	0	36	DIVF	FP_DIV
Floating point Control	1	1	0	0	0	1	49	JEQF	FP_BRANCH
	1	1	0	0	1	0	50	JNEF	FP_BRANCH
	1	1	0	0	1	1	51	JGTF	FP_BRANCH
	1	1	0	1	0	0	52	JLTF	FP_BRANCH
	1	1	0	1	0	1	53	JGEF	FP_BRANCH
	1	1	0	1	1	0	54	JLEF	FP_BRANCH
Trap	0	1	1	1	1	1	31	TRAP	TRAPS

Table A.1: Opcode table

A.2.2 Data Types

PERL architecture supports 8 types of integer data and two floating point data types as shown in table A.2.

A.2.3 Addressing Modes

PERL architecture supports 4 addressing modes as shown in table A.3. If addressing mode indicator bits are 00, it indicates base addressing mode for arithmetic and logical instructions, and PC relative addressing for branch instructions. If an immediate mode is used for destination the results are discarded.

In the example below the dest, src1 and src2 are respectively represented using direct, base and indirect addressing modes.

```
add _p:b1,200(SP):b2,(_size8):b4
```

Data Type	Value(3 bits)	mnemonic
unsigned byte1(8 bits)	0	ub1
unsigned byte2(16 bits)	1	ub2
unsigned byte4(32 bits)	2	ub4
unsigned byte8(64 bits)	3	ub8
signed byte1(8 bits)/float4(Single Precision FP)	4	b1/fp4
signed byte2(16 bits)/float8(Double Precision FP)	5	b2/fp8
signed byte4(32 bits)	6	b4
signed byte8(64 bits)	7	b8

Table A.2: Data types of operands

Addressing Mode	Value (2 bits)
Base/PC relative	0
Direct	1
Memory Indirect	2
Immediate	3

Table A.3: Addressing modes for operands

The three memory operands can be specified independently using any of the 4 addressing modes. Further, the data type of each of them can also be independently specified. The base addresses are specified using only 2 bits. These refer to one of the four addresses stored in fixed locations and are cached permanently.

Appendix B

Simulator Configuration File

To analyze the effect of various parameter values on processor performance, the simulators for PERL and DLX read a configuration file. Values for various different parameters can be specified in this file. In this appendix, we describe the format of this file.

The simulator has a set of default values for all these parameters and it is forced to pick up the parameter values from this configuration file by invoking the simulator with a command line argument as shown below (md is the configuration file name).

```
# supersim -f md
```

B.1 Instructions Process per Cycle

The number of instructions that needs to be fetched, decoded, issued and committed per cycle during simulation is specified as shown below. The first field indicates the operation and the second field specifies the value for it.

```
Instructions_process_per_cycle
fetch           4
decode          4
issue           4
commit          4
```

B.2 Memory

The size of simulator's memory in bytes, the latency in terms of number of cycles and the maximum number of memory accesses (L1 cache) per cycle are specified as shown below. The maximum number of memory accesses is essentially the number of ports in L1 data cache.

```
Memory
size            19000000
latency         1
accesses        2
```

B.3 Reorder Buffer Size

The maximum number of entries in each of the integer and floating point reorder buffers are specified as shown below.

```
Reorder_Buffer_size
integer         40
float           40
```

B.4 Instruction Window Size

The maximum number of entries in each of the integer and floating point instruction windows are specified as shown below.

```
Instruction_Window_size
integer         40
float           40
```

B.5 Instruction Queue Size

The maximum number of entries in instruction queue is specified as shown below.

```
Instruction_Queue_size  40
```

B.6 Branch Buffer Size

The maximum number of entries in the branch target buffer (BTB)is specified as shown below.

```
Branch_Buffer_size      111
```

B.7 Integer Functional Units

The number of each of the integer functional units ALU, shift, branch, multiply and divide and their respective latencies in (number of cycles per operation), are specified as shown below.

```
Integer_Functional_Units
alu
number          4
latency         1
shift
number          2
latency         1
branch
number          1
latency         1
mult
number          2
latency         5
div
number          2
latency         10
```

B.8 Floating Point Functional Units

The number of each of the integer functional units ALU, shift, branch, multiply and divide and their respective latencies are specified as shown here.

```
Floating_Point_Units
add
number          4
latency         2
mult
number          4
latency         5
div
number          2
latency         10
address
number          4
latency         1
branch
number          1
latency         1
```

Appendix C

Cache Simulator Input Files format

C.1 Trace File

The *dinero* format is a very simple and effective format to capture the memory address trace. In *dinero* format every memory reference is characterized in a 3-tuple written in a single line as shown below. The first field is an integer representing the reference type, the second field specifies the memory address in hexadecimal format and the third field specifies the clock cycle at which the processor generated this reference. There are three types of memory references: memory read (0), memory write (1) and instruction fetch (2).

```
#reference type          memory address(in hexadecimal)  clock-cycle
   2                          100                            1
   2                          104                            1
   1                          4af                            5
   1                          4af                            5
   0                          4ff                            7
   0                          4f0                            7
```

C.2 Cache Configuration File

The cache simulator will simulate the cache whose organization is specified in the configuration file (*cache.config*). The format of *cache.config* is explained below.

C.2.1 Levels

The first parameter that has to be specified is the number of cache level to be simulated. The first field in the file specifies this parameter.

C.2.2 Cache Specification of Each Level

Depending on the number of cache levels specified, the user has to specify the parameters for each levels of cache starting from the highest level of cache. Comments can be inserted by placing a `#` symbol as the first character in the line.

For each level the first parameter is the type of cache, which can be either unified (0) or split(1). This is followed by the number of blocks, block size in bytes, the associativity, number of interleaved ports and number of duplicate ports. This is followed by the write policy (write-back is specified by 0 and write-through by 1). The last parameter is the miss penalty for this level of cache.

In case the cache is of split type, the instruction cache specification has to be specified first. Since instruction cache blocks are read only, the write policy is not specified. However, the miss penalty, which would be common for both instruction and data cache, is specified at the end of data cache specification of that level.

A sample specification is given below. It specifies a 2 level cache. The second level cache is a unified cache with 1024 blocks and has a block size of 64 bytes. It is a 4-way set associative. It has a single port, employs write back policy and has a miss penalty of 10 cycles.

The first level cache is a split type. The instruction cache has 256 blocks with a block size of 64 bytes. It is a direct mapped cache (1-way associative) and has a single port. The data cache has 256 blocks with a block size of 32 and is direct mapped. It has 2 interleaved ports, 2 duplicate ports and employs a write-through policy with a miss penalty of 4 cycles.

```
#Number of Cache levels
2
#Level 2 cache specification
#Type of cache     Number of blocks  Block size(bytes)  Associativity
0                      1024                64                4
#Number of Interleaved ports  Duplicate Ports   Write Policy
          1                        1            0 (write back)
#miss penalty for this level
         10
#Level 1 cache specification
#Type of cache
1
#Instruction cache specification
#Number of Blocks  Block size  Associativity
    256              64           1
#Number of Interleaved ports   Duplicate Ports
            1                      1

#Data Cache specification
#Number of Blocks  Block size  Associativity
      512             32          1
#Number of Interleaved ports   Duplicate Ports    Write Policy
         2                        2             1 (write through)
#miss penalty for this level
         4
```