Evaluation of a Scalable Decoupled Microprocessor Design

By

GARY SCOTT TYSON
B.S. (California State University, Sacramento) 1986
M.S. (California State University, Sacramento) 1988

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA

DAVIS

This is a reformated version of the dissertation to reduce the number of pages required to print it from 238 to 104. It is unchanged from the original with the following exceptions:

- Modification of title page.
- Removal of acknowledgements page.
- Line spacing changed from double spaced to single spaced.
- Appendix A condensed from 76 pages to 1 page.

Evaluation of a Scalable Decoupled Microprocessor Design

Gary Scott Tyson

Doctor of Philosophy in Computer Science
University of California, Davis
***Professor Matthew K. Farrens*, Committee Chair**

Abstract

In this dissertation a new architecture is described and analyzed to determine its capability to extract instruction level parallelism from applications written in the C programming language. The architecture is novel in its use of multiple independent instructions streams to exploit the very fine grained parallelism found in these applications.

The research consists of three parts: the development of the new architecture, the development of a compiler model to translate the source program into machine code for that architecture and the development of a new cache structure capable of satisfying data requests at a sufficiently high rate to feed the processor without performance degradation.
Keywords: decoupled architecture, instruction level parallelism, compiler design, cache organization.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In order to satisfy the ever-growing computational requirements of computer users, computer manufacturers continue to strive for faster and faster designs. Over the years, many different approaches have been used to improve processor performance. For example, CRAY-class computers focus on driving up the clock frequency, while *Complex Instruction Set Computer* (CISC) machines attempt to reduce the number of instructions required to complete a task. Most current single-chip processors try to both maximize the clock frequency and minimize the average clock cycles required per instruction (CPI).

While increases in clock frequency continue to drive the improvement in processor performance, chip designers have also looked to other techniques to improve performance. For example, instruction throughput can be increased by locating instructions that have no dependent relationship and executing those instructions in parallel. This parallelism comes in two forms: Instruction Level Parallelism (ILP) and data parallelism. By exploiting this parallelism, the CPI term can be reduced dramatically.

1.1 Finding Parallelism in a Program

1.1.1 Instruction Pipelining

Instruction pipelining is a processor implementation technique which separates the execution of an instruction into distinct pipeline *stages* and allows one or more instructions to be in each stage during any clock cycle. The ability to support multiple instructions in the pipeline is possible because each pipeline stage requires different processor resources The independence of pipeline stages makes it possible to complete one instruction completion per cycle. As chip designers look to continue the increase clock frequency, there is a corresponding increase in pressure to reduce the number of gates traversed during each cycle, and thereby require more pipeline stages to execute an instruction. Increasing the number of pipeline stages can exacerbate the difficulty in finding independent instructions to fill those pipeline slots. This limits the ability to achieve improved performance through pipelining alone; further mechanisms must be explored to identify and exploit additional parallelism to obtain more performance.

1.1.2 Instruction Level Parallelism

Instruction level parallelism is a measure of the number of instructions that can be issued during a single clock cycle [1]. These instructions must not compete for the same resources and must not have a dependence (e.g. a value generated by one instruction and used by another). Independent operations exist because of the characteristics of the programming model, which separates a task into a number of different subtasks (execution control flow, memory I/O activity, data manipulation, etc). In many cases these operations are completely independent (e.g. the calculation of an address used to fetch data often has nothing to do with the eventual manipulation of that data) and are therefore prime candidates for executing in parallel.

1.1.3 Data Parallelism

Data parallelism, in contrast to ILP, is a property of the task itself. Some tasks contain data manipulations that are completely independent from one another. These tasks can be thought of as containing explicit parallelism that can be exploited regardless of the programming model employed. Many architectures exist which exploit data parallelism, including vector processors [1], *Single Instruction Multiple Data* (SIMD) [2] designs, *Multiple Instruction Multiple Data* (MIMD) [3] designs, and dataflow machines [4].

The key difference between instruction level parallelism and data parallelism is the difference in locality of the dependencies between adjacent instructions; instruction level parallelism exists when instructions are independent from some adjacent instructions but may have dependencies with others that are close in the execution stream. Allowing dependencies between close instructions requires the ability to support frequent bi-directional communication between operational units. This high level of communication is often unnecessary in data parallel applications where the task can be decomposed into far more independent pieces.

[1] Instruction issue refers to the act of assigning an instruction to a functional unit. A common four stage pipeline design consists of *instruction fetch*, *decode*, *execute* and *write-back* stages with instructions being issued to the *execute* stage.

1.2 Exploiting parallelism using multiple instruction streams

Extracting parallelism on a MIMD architecture has traditionally been accomplished by partitioning a program into data independent portions and assigning them to separate processing elements, ignoring any other parallelism that might exist. Examples of this type of architecture include the MIT Alewife machine [6], the Stanford DASH architecture [7], and the Wisconsin Windtunnel design [8]. The separation of a program into multiple single issue instruction streams allows the decentralization of the hardware resources in these architectures by replacing a central instruction window with multiple windows from which instructions can be issued. Furthermore, each of these instruction windows can be less complex. The register file can also be distributed among all processors in a MIMD design reducing register access contention. This eliminates the need for complex multi-access memory cells (required by a centralized register file) providing greater expandability.

While a MIMD approach to code scheduling clearly possesses certain advantages, historically these architectures have suffered from severe limitations. Data transfer latencies have been high, and the bandwidth required to support high-throughput, low contention data transfer between processors has been unavailable because of pin and/or board-level interconnect limits. In addition, it is often necessary to synchronize the different instruction streams in order to ensure program correctness. Including these synchronization points can cause unacceptable performance loss. Using main memory to handle data transfers between processors can also lead to an unacceptable dependence on memory latency. These problems help explain why current MIMD designs do not attempt to exploit ILP.

Both bandwidth and latency limitations in MIMD designs can be overcome if sufficient resources can be allocated to place all of the processors on the same chip. Increasing the number of transistors that can be fabricated per square centimeter provides a means by which many of the interprocessor communication problems can be eliminated. Placing several processing elements on the same die circumvents the pin limitations on bandwidth, and supports high on-chip data transfer rates. In addition, using First-In-First-Out (FIFO) ordered queues in a manner similar to that used by decoupled machines [9] provides a clean way to handle synchronization.

As transistor densities have continued to increase, single chip MIMD designs are now becoming feasible. One study [10] indicates that as tens of millions of transistors become available, something more than simply increasing on-chip cache sizes must be done. This observation led to the design of the *Multiple Instruction Stream Computer* (MISC) architecture, a decoupled MIMD machine that is designed to support and exploit instruction level parallelism [11].

1.3 A Multiple Instruction Stream Computer

The MISC architecture was designed to study a wide range of design points in exploiting all levels of parallel program execution. To enable this study, the MISC design incorporates many of the characteristics of shared memory and message passing architectures, as well as the capabilities of previous decoupled and other ILP designs. It was hoped that this approach would exploit both the instruction and data parallelism available in a task by combining the capabilities of traditional data parallel architectures with those found in machines designed to exploit instruction level parallelism. For example, MISC is designed to support multiple instruction issue without increasing the complexity of instruction issue logic or affecting clock frequency. Furthermore, the regularity and simplicity of the MISC design should enable a shorter design cycle and increased scalability.

1.4 The Compiler's Role

The role of the compiler in an ILP processor differs widely depending on the type of processor. A superscalar processor requires no special help from the compiler specifying the parallelism available — the hardware is responsible for identifying any independent operations that can be issued on the same clock cycle. At the other extreme, a VLIW compiler is responsible for generating a complete specification of the parallelism in the program — the hardware simply executes the instructions in the order specified in the schedule.

There is a performance tradeoff (in both execution and design time) between compiler (static) and hardware (dynamic) scheduling; the more information the compiler can convey to the hardware, the simpler the hardware design can be. This leads not only to shorter design time, but also to higher clock frequencies. However, requiring the compiler to completely specify the available parallelism can limit the portability of the generated code; even if alternate implementations can execute a given binary, a severe performance degradation may result.

A second problem with the current architectural model used during compilation is the extensive use of centralized mechanisms to distribute the results of computations. For example, a centralized register file is assumed which links dependent operations from all processing resources. This limits the scalability of these designs.

To eliminate these deficiencies a more flexible, decoupled architectural model is proposed. The idea is to reduce the complexity of the implementation by increasing the ability of the programmer (or compiler) to specify the parameters of execution. This leads to a design which has different goals than most current ILP machines. These design goals include decentralizing the resource requirements for each phase of the instruction pipeline, the ability to exploit the locality found in program execution, simple RISC-like implementation, and an object format that allows the compiler to convey more information about instruction order and resource allocation than current ILP designs convey.

1.5 Contribution of this Dissertation

This thesis contains a number of contributions to the field of computer architecture:

- A new architecture is proposed which utilizes a more scalable decentralized instruction fetch mechanism than found in current processors, a simplified decode pipeline stage, a decentralized register file and a distributed internal processor clock.
- The design and construction of a compiler capable of distributing general C code across multiple asynchronous processors in a manner that enables greater tolerance of high operational latencies.
- Experimental analysis of the effectiveness of exploiting instruction level parallelism by employing multiple program counters in a MIMD architecture. It will be demonstrated that the MISC system is capable of achieving high performance execution comparable to that of current superscalar designs.
- Memory system enhancements supporting out of order memory operations and improved cache management. These enhancements are shown to improve cache hit rates beyond that of current approaches and are equally applicable to MISC and superscalar processor designs..

1.6 Organization of this Dissertation

The remainder of this dissertation is organized into six chapters as follows:

- Chapter 2 provides a more detailed discussion of ILP and describes existing approaches used to extract that parallelism.
- Chapter 3 introduces a new multiple instruction stream architecture capable of exploiting ILP without the necessity of a centralized clock or centralized resources.
- Chapter 4 describes the compiler constructed to translate C source code to the MISC architecture.
- Chapter 5 analyzes the effectiveness of the MISC architecture in exploiting parallelism in two sets of benchmark applications traditionally used to measure the performance of superscalar, VLIW and other high performance, ILP architectures.

- Chapter 6 examines the performance of the MISC memory interface, comparing an improved interleaved cache approach to that of a multi-ported cache design.

- Chapter 7 concludes this dissertation and describes the future research potential of the MISC approach for improving processor performance.

- Appendix A gives a description of the MISC instruction set, including a detailed description of each MISC instruction.

Chapter 2

Instruction Level Parallelism

The amount of instruction level parallelism in a program is a measure of the number of instructions that can be issued during the same cycle and is a function of the dependencies and operational latencies of the program. Several different types of dependencies can exist in a program as implemented on a certain architecture: These can be broadly categorized as *data* and *control* dependencies. A data dependency exists when data which is generated by one instruction is used by a subsequent instruction; since the data use (read) occurs after the definition (write), this is referred to as a *Read After Write* (RAW) or true dependence. Additional dependencies can also occur due to the reuse of processor resources, such as a register re-definition. For instance, if an instruction uses (reads) a register which is then re-defined (overwritten) by a later instruction, a dependency exists preventing the simultaneous execution of both instructions; this is referred to as a *Write after Read* (WAR) or anti-dependence. Similarly, a register definition followed by a second definition creates a resource dependency between the instructions and is referred to as a *Write after Write* (WAW) or output dependence.

Control dependencies exists at control flow points in the execution. When a conditional branch instruction is executed, for example, all instructions following the branch are dependent on the branch outcome. These dependencies can severely limit the available parallelism in most applications because of the high frequency of branch instructions.

2.1 Extending Issue Widths in Current Architectures

The early 1980s saw an emergence of architectures designed to support multiple instruction issue at each clock cycle. Several companies (Multiflow, Cydrome, Culler) built multiple issue architectures that incorporated large instructions and multiple ALU operations [12] [13]. These *Very Large Instruction Word* (VLIW) processors were capable of supporting much larger amounts of parallelism for scientific and engineering codes than previous, single issue architectures. While each of these companies eventually failed as a business venture, the ideas and compiler techniques they developed have found acceptance in most of the high performance designs of today.

Superscalar architectures appeared shortly after the original VLIW designs and started shipping in the mid 1980s. These systems combined the multiple-issue execution pipeline used in VLIW designs with the sequential instruction set architecture found in previous scalar processors, allowing superscalar implementations to execute existing object code much faster. Some of the earliest superscalar processors came from Apollo Computers [14], IBM [15] and Intel [16].

Decoupled processor designs also began to appear in the 1980's both in supercomputer [9] and microprocessor [17] versions. These processors separate program execution into distinct *tasks*, each of which executes instructions from an independent stream. For the most part decoupled designs have been relegated to a university research setting [1].

Various taxonomies have been proposed that identify the important distinctions between VLIW and superscalar architectures. The most useful of these are based on the interaction between the compiler and hardware in scheduling the code. To develop their taxonomy, Rau and Fisher [18] examined the various ways in which the compiler and hardware can cooperate in locating the ILP available in an application.

Figure 2.1 is a graphic representation of the relationship between the compiler and hardware in locating instruction level parallelism for VLIW, superscalar and decoupled processor designs. The division in responsibility between the compiler and the hardware can be viewed in two stages:

1. Determine whether a given operation is dependent or independent of those yet to be issued.

2. Bind the resources necessary for independent operations to execute at some particular time, on some functional unit with specific source and destination locations.

Looking at Figure 2.1, we see that the program specification for a *Sequential* architecture requires no explicit information about dependencies; dependency information is conveyed through the strict sequential ordering of instructions in the object format. Each stage of extracting parallelism is the responsibility of the hardware. These architectures require complex instruction windows to reschedule

[1]Both VLIW and decoupled architectures have only rarely been developed as commercial products because they lack the ability to execute existing executable programs. Superscalar designs have been chosen by almost all processor manufacturers because of their ability to execute applications previously compiled for scalar version of an architecture.

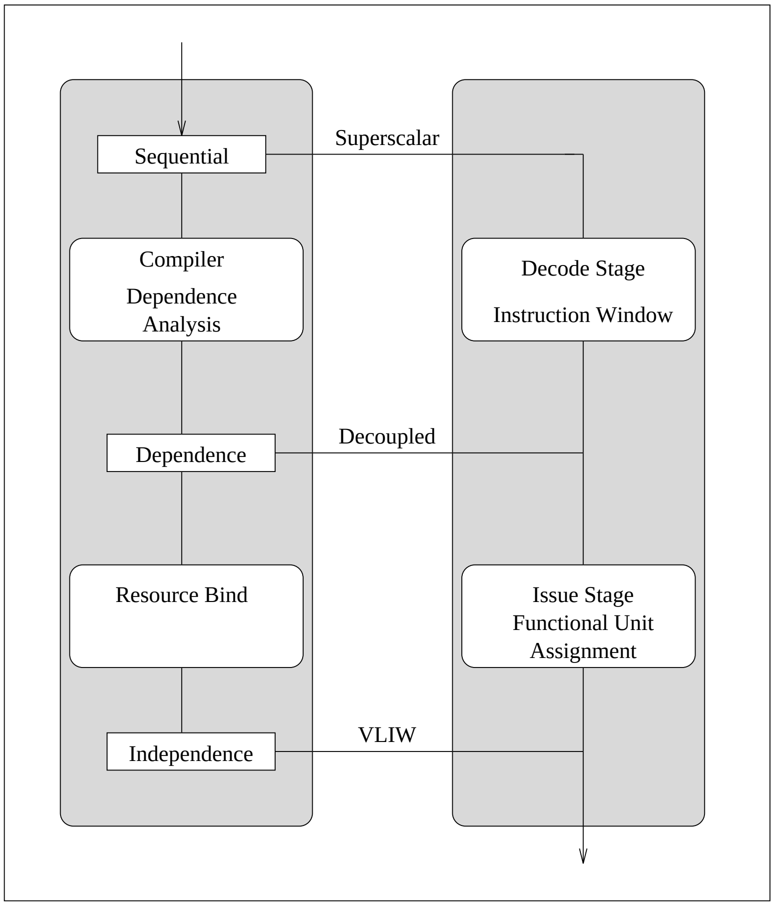

Figure 2.1: ILP Taxonomy: Division of responsibility between the compiler and the hardware

instructions in a manner more optimal to a particular configuration. Unfortunately, this centralized instruction decode/window can affect the clock speed and limit scalability [2] of the design. It becomes very difficult to design a large instruction window (required to locate a large number of independent operations) which is fast enough to not affect the processor cycle time. Superscalar processors fall into this category.

A second method of interaction between the compiler and architecture is required for an *Independence* architecture. In this approach, the compiler is responsible for completely specifying how all resources will be allocated on the implementation. Independence architectures, such as VLIW, require compiler support for determining all aspects of parallel execution. Unfortunately, by requiring the compiler to completely specify the binding of instruction to functional units, code portability is lost. Any new implementation of an architecture (e.g. increasing the

[2] Scalability in the context of high performance ILP designs exists when more than 4-8 instructions can be issued each cycle.

size of the cache or the latency to main memory) requires re-compilation of the program to account for the new resource constraints.

A *Dependence* architecture, on the other hand, only requires the compiler to *convey information* about both instruction dependencies and independencies in order to exploit parallelism; the actual binding of instructions to functional units is still performed dynamically. Decoupled architectures specify dependencies by assigning instructions to independent instruction streams and explicitly specifying any dependent operations through a transfer via an architectural register queue. Dependence architectures avoid the requirement for a complex decode stage and instruction window, while allowing dynamic control of instruction issue and functional unit allocation.

2.2 Existing Machines

The first commercially successful machines that provided a modicum of instruction level parallelism appeared in the 1960s. They extended earlier architectures by including multiple function units, allowing multiple computations to execute simultaneously.

2.2.1 Early ILP Architectures

In 1963 Control Data Corporation completed the CDC 6600 [19] which had 10 functional units and could start execution on any unit independent of the execution state of the other units. The hardware was responsible for determining where and when an instruction should be executed, using a technique referred to as *scoreboarding* [20]. Scoreboarding is a centralized control technique that performs the bookkeeping operations necessary to allow out-of-order execution [3]. In the scoreboard, data requirements of an instruction are examined, identifying those instructions which have all source operands available and the desired functional unit ready. The scoreboard communicates with the functional units in order to control each instruction's progress from the issue stage of the pipeline until completion. Unfortunately, by utilizing a central structure to control instruction flow through the pipeline, scoreboarding does not scale well.

A later system, introduced by IBM, addressed some of the deficiencies of the CDC 6600. The 360/91 machine, introduced in 1967, had fewer execution units than the CDC 6600 but used a more aggressive instruction issue policy in order to maximize the utilization of the execution units. The 360/91 used a decentralized instruction flow control algorithm developed by Tomasulo [21], which uses a number

[3] Out-of-order execution describes the function of a pipeline implementation which allows instruction to be executed in a different order than the sequential program order specified in the object code.

of buffers associated with each functional unit and a structure called the *common data bus*. These buffers, called *reservation stations*, serve to control instruction issue by holding an instruction until all of the source operand values are available and the functional unit is ready.

These early ILP architectures were limited to issuing at most one instruction per cycle. Most current ILP designs extend the capabilities of one of these systems by increasing the size of the buffers and the overall width of the pipeline (i.e. the number of instructions that can be in any pipeline stage at one time).

2.2.2 VLIW Architectures

Since the compiler has the most complete information about the entire program, it is well suited to deal with the inclusion of additional resources (e.g. ALUs, FPUs and I/O units), and can often increase instruction execution bandwidth in areas of the code that were previously performance limited by resource constraints. A *Very Long Instruction Word (VLIW)* machine can exploit more parallelism by increasing the computational resources (e.g. ALUs) and encoding the function for each of these resources in a compound instruction word; this allows multiple operations to be initiated when one of these compound instructions is issued. The instruction word contains an opcode field for each functional unit; this simplifies the initial pipeline stages (*fetch, decode and issue*) because the compiler specifies which operations can be performed during any clock cycle. In a VLIW architecture, it is up to the compiler to explicitly schedule the use of each operational unit in the processor, placing independent operations in the same compound instruction word. No-op operations are assigned to functional units for which no independent instruction can be located [4]. The static placement of instructions required in a VLIW design does not support the dynamic reordering of operations by the hardware (i.e. it does not support out-of-order execution). Furthermore, any change of the hardware description requires all code to be recompiled in order for the program to work correctly. Very recent work has attempted to remove this constraint by rescheduling the machine language program directly.

2.2.3 Superscalar Architectures

While VLIW architectures can efficiently exploit parallelism found in many applications, doing so requires the re-compilation of the original source representation of that application. *Superscalar* architectures, on the other hand, employ a hardware scheduler that uses dynamic run-time information in order to efficiently allocate resources to the list of instructions ready for execution. This allows superscalar implementations of existing architectures to execute previously compiled programs;

[4] VLIW architectures derive their name from their need for a large instruction word to specify the task of each functional unit.

a tremendous advantage when the software base of an existing architecture is distributed in compiled (binary) form, as in the case of the x86 line of processors. When previously compiled (*legacy*) codes account for a majority of the applications executed, a superscalar approach becomes very attractive despite its inability to exploit as much parallelism as alternate approaches can.

Superscalar implementations provide the flexibility of separating implementation details from the architectural specification by incorporating a hardware scheduler to dynamically reorder instructions. However, this scheduler, which selects instructions from a fixed-size window of available instructions, does not have access to the breadth of information available to the compiler. This eliminates the ability of these processors to exploit parallelism that is not located within the instruction window.

2.2.4 Decoupled Architectures

A third approach to issuing multiple instructions per cycle takes advantage of the characteristics found in the Von Neumann computational model. *Decoupled* architectures attempt to exploit the independent nature of control flow, memory access and data manipulation operations that comprise conventional computations by splitting a program into distinct tasks and executing each task on separate pieces of hardware. These hardware units communicate data and control information via FIFO queues so the instruction streams are not required to execute in lock-step. This means that the inability of one PE to execute instructions does not affect the ability to execute instructions on the other PEs — thus providing dynamic support for out-of-order execution. This approach is designed to take advantage of the best that both VLIW and superscalar have to offer; the compiler partitions the tasks in a manner similar to VLIW, and the queues provide the same dynamic scheduling benefits found in superscalar.

Decoupled systems differ from VLIW and superscalar designs in the manner in which the independently issued instructions interact. VLIW and superscalar processors can be thought of as very tightly coupled shared memory systems; they share not only addressable memory but also register space. This shared register approach differs from the explicit message passing (via FIFO ordered queues) found in decoupled machines. Furthermore, in order to transmit data among operational units by writing and then reading the contents of a register, the clocks on VLIW and superscalar processors must be synchronized. This requirement is relaxed with an explicit message passing approach.

The greater flexibility found in a decoupled design allows both single and multiple instruction stream descriptions of a task. The *ZS-1* [9] and *WM* [22] systems operate in a decoupled manner while receiving instructions from a single instruction stream. Their architectural component descriptions are similar to those of *Split Register* superscalar designs [23].

The *PIPE* machine [24], in contrast, consists of two PIPE processors [25] which run asynchronously, each with their own instruction stream, and cooperate on the execution of a single task. The PIPE processor uses branch queues and a BFQ, *Branch From Queue*, instruction to coordinate the outcome of branch decisions between processors [26]. By using homogeneous processors the PIPE machine has the ability to execute in either decoupled access/execute mode or in a single processor (SP) mode. In SP mode both processors are used, but they execute independent processes. PIPE was one of the first decoupled architectures to be implemented [27].

2.3 Previous Decoupled Compilers

Several compilers for decoupled machines have been developed. These include the original PIPE compiler [28], the WM *streams* compiler [22], and the compiler for the Briarcliff Multiprocessor [29]. These compilers differ from those for conventional processor designs in the explicit use of architecturally visible register queues. In addition, the PIPE and Briarcliff compilers were responsible for partitioning the program into separate instruction streams to be executed on individual processing elements (connected via communication queues and a shared addressable memory).

2.3.1 The PIPE Compiler

The PIPE compiler separates code into *access* and *execute* instruction streams. This is accomplished by assigning each branch and memory access operation to the *access* processor, then examining a Program Dependence Graph (PDG) [30] to determine which additional branch control and address calculation operations should also be assigned to that processor. All remaining instructions, as well as duplicate branch operations, are then given to the *execute* processor. Once this separation is accomplished, register allocation and other optimization transformations can be applied to each instruction stream.

2.3.2 The WM Compiler

The WM compiler is more conventional in its use of a single instruction stream. Dataflow analysis and many of the optimization transformations performed are unchanged from standard scalar designs, with additional restrictions placed on the register allocation method to account for the nature of the memory queues found in this decoupled architecture.

2.3.3 The Briarcliff Compiler

The compiler used in the Briarcliff Multiprocessor performs in a much different manner than the previous two compilers. This compiler is far more aggressive in separating code into multiple instruction streams. Instructions are partitioned equally over the available processing elements (PEs) [5], with those data dependencies that exist between PEs being allocated a register channel [31]. Optimization is then performed to reduce the number of channels required without degrading code performance. Memory operations can also be performed on register channels. This means that memory accesses can originate in one PE (which calculates the effective address) with the data either coming from or destined for different PE.

The Briarcliff design bears more of a resemblance to a VLIW architecture than a decoupled architecture in its treatment of control flow operations. PEs synchronize on branch operations by generating a global condition code used to determine whether or not to branch. While each PE may reach the actual branch instruction on different cycles, each branch point serves as a barrier synchronization point; no PE is allowed to process instructions past that branch until each PE has completed its branch decision. This *fuzzy barrier* [29] mechanism allows more flexibility than a VLIW implementation but fails to provide the flexibility found in true MIMD approaches like PIPE and MISC.

2.4 Summary

Exploiting instruction level parallelism on MIMD architectures has the potential to overcome both the complexity required by superscalar pipelines and the rigid execution framework of VLIW processors. The instruction issue stage of each processor in a MIMD design can perform in a simple single-issue, in-order manner, avoiding much of the hardware complexity required to support out-of-order issue in a single instruction stream approach. Out-of-order issue is also supported on a MIMD because the processors are run independently; therefore, any independent instructions executed on different processors can issue in any order without necessitating extensive hardware support. This is fundamentally different than multiple issue in a VLIW machine, because a strict ordering of instructions is not imposed by the compiler unless a dependence exists. Furthermore, by incorporating multiple program counters, a MIMD machine provides the architecture with more dataflow information by enriching the specification of the object language; taken to its extreme this would allow a dataflow machine description of the program.

[5] Throughout this dissertation a processing element is defined as an execution unit complete with a program counter (PC).

Chapter 3

Design of the MISC Processor

The Multiple Instruction Stream Computer (MISC) system is a new architecture that has been developed at the University of California, Davis to study the characteristics of exploiting ILP on a MIMD processor design. MISC is composed of multiple Processing Elements (PEs) which cooperate in the execution of a task, coordinating through a message passing system allowing data to be transferred between PEs as easily as it can be placed in a register.

This chapter will describe the design of the MISC architecture. The design goals for the MISC processor will be presented, followed by a detailed description of each component of the design. Those portions of the design that mirror conventional uni-processor design (e.g. pipeline structure) will only be mentioned briefly in order to focus the discussion on the unique capabilities found in this architecture. Many of these capabilities require compiler support and are described in further detail in later chapters.

3.1 Design Goals

The MISC architecture is designed to study the feasibility of separating the execution of an application into multiple instruction streams. There were several design goals for the MISC processor:

1. Decentralize most, if not all, processor resources. This is considered desirable because centralized resources (such as a heavily ported register file) can affect the maximum clock frequency attainable and reduce scalability.

2. Support compiler directed out of order execution. This leads to a reduction in the complexity of the instruction issue mechanism by eliminating the need for dynamic reconstruction of dependence information. An overly complex instruction issue design can both slow clock frequency and increase processor development time. In essence, the goal is to enable the compiler to convey

information to the hardware about instruction dependence, eliminating the need to re-construct these dependencies dynamically. This reduces the complexity of the implementation while providing a richer interface between the compiler and the implementation of an instruction set architecture.

3. Develop a high performance memory system capable of supporting multiple memory accesses during each clock cycle. Two desirable characteristics of a memory system are the ability to reorder non-conflicting memory operations (those referencing different addresses) and the ability to handle multiple references per cycle. This is accomplished in MISC by allowing the compiler to specify a partial ordering of memory operations instead of the complete order imposed by conventional scalar (and superscalar) architectures.

The following sections describe the MISC component structure, the design of the processing elements and the interface to the memory system through the cache.

3.2 MISC Component Structure

A MISC processor may contain any number of processing elements. In a single PE configuration, MISC differs from conventional architectures only in its use of register and memory queues. In a dual PE configuration, MISC operates like a PIPE machine. Configurations with more PEs allow more flexibility in assigning operations to individual PEs.

The MISC system described in this chapter consists of four PEs, a two-way interleaved, unified cache (UCache) and a set of internal data paths used to transmit data among PEs and the UCache [1]. All PEs can send information to any other element or collection of elements, while the UCache can send data to any PE. This configuration will be identified as $MISC_4$ throughout the remainder of this dissertation and will be used to illustrate characteristics of decoupled code partitioning discussed in the next chapter. The component design of MISC is illustrated in Figure 3.1. The three primary components in the $MISC_4$ processor are:

- The set of processing elements [PE1 through PE4].
- a unified cache which serves both as a high throughput cache and as the interface with the memory system [UCache].

[1] A four PE version was chosen to illustrate the features of the MISC design because it has enough processing elements to make partitioning code across all elements a difficult task, without being so large as to make the example codes too complex to convey information easily. Each of the tools, e.g. the compiler and simulator, are capable of supporting both larger and smaller configurations. The compiler will be described in the next chapter and the simulator will be discussed in chapter 5.

- a set of internal data busses connecting each processing element as well as the unified cache [BUS1 through BUS4, CBUS1 and CBUS2]

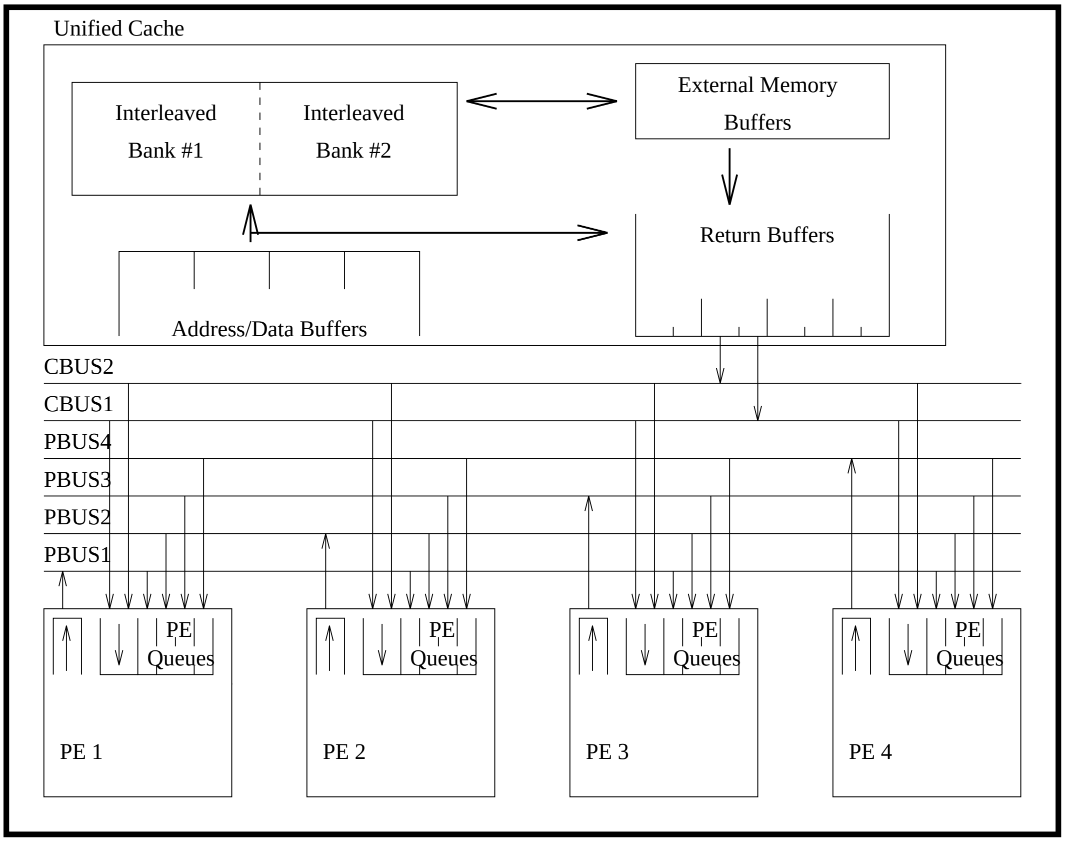

Figure 3.1: $MISC_4$ Component Design

3.3 MISC Internal Bus Structure

In $MISC_4$ each PE employs an independent clock; communication between PEs or between a PE and the UCache utilizes the internal datapaths and proceeds in an asynchronous manner. Each data path is controlled by a single element; for instance, the internal data path labeled PBUS1 is controlled (written to) solely by PE1. Each PE has its own bus (PBUS1-4), and the UCache controls two other busses (CBUS1 and CBUS2).

Each internal bus consists of 32 *data* lines, 5 *routing* lines and 5 *busy* lines. The *routing* lines are used to identify the destination of a given message; a *routing* line exists for each possible destination (PE1-4 and UCache). Collectively, the *routing* lines specify which subset of destinations should receive a message. In order to broadcast a message to all other processors, a PE or UCache asserts the *routing* line for each destination, enabling up to 5 data transfers in a single PBUS or CBUS operation. The *busy* lines are asserted by each potential destination to indicate its inability to accept more data at the present time.

When a processing element wants to send a message, it determines the desired *route* and compares this with the appropriate *busy* lines. If all destinations are capable of receiving the data, then the data and routing information is placed onto the bus [2]. Transmission of data does not require the communicating PEs to synchronize at the transfer of the message; instead, data buffers (queues) are provided to store messages that have been sent but not yet processed by the destination. This allows the PE that originates a transfer to continue with processing while the transfer itself is delayed until all recipients are ready [3].

3.4 Processor Structure

$MISC_4$ PEs each have their own independent instruction streams. Each PE maintains all state information required to function as an independent processor. In fact, the $MISC_4$ hardware is capable of running four completely unrelated tasks in parallel. However, this dissertation will focus on partitioning a single task across four independent instruction streams.

Homogeneous PEs are modeled in order to allow the compiler maximum flexibility in assigning operations. This also simplifies the design of the system as a whole since only a single PE needs to be created [4]. Figure 3.2 shows the structure of a processing element.

Each PE employs a basic four-step execution pipeline, including single cycle Instruction Fetch, Instruction Decode, and Instruction Issue phases and a multi-cycle Execution phase. Each of the processing elements (PEs) contains the following components:

- A set of General Purpose Registers (GPR's)
- A set of Intra-PE Data Queues (PEQ's)
- Two Memory Data Queues (MQ's)
- An Integer Arithmetic/Logic Unit (ALU)

[2] If any destination queue is busy, then transmission is delayed until all recipients are ready to receive the data.

[3] The size of each queue can be assigned independently. Generally a small queue size (2-5 elements) is all that is necessary between PEs, while a a somewhat larger queue size is required to hide the larger latency of memory loads (5-20 elements).

[4] Allowing a heterogeneous mix of PEs may allow for further PE optimization to be performed at the hardware and compiler level; this approach will be discussed in more detail in chapters 5 and 7.

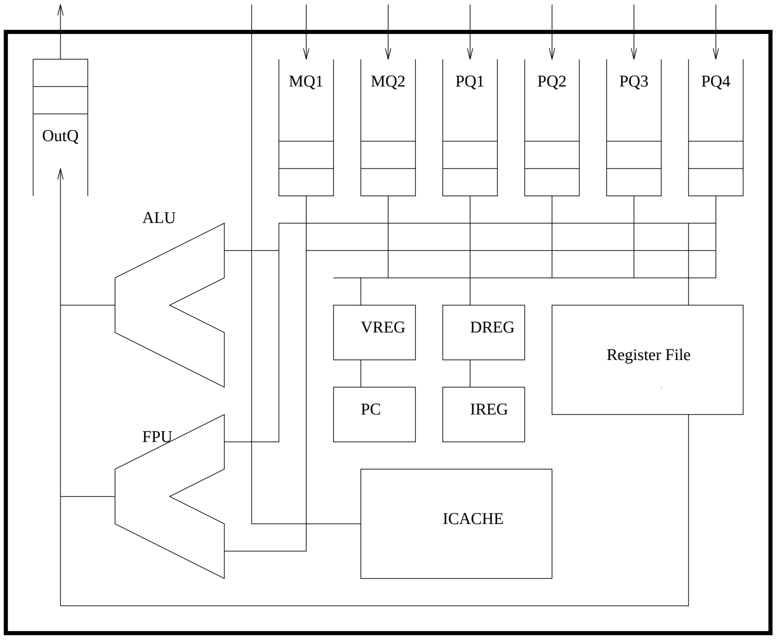

Figure 3.2: MISC Processing Element

- A Floating Point Arithmetic Unit (FPU)
- A Program Counter (PC)
- A Vector Register (VREG)
- An Output Queue (OutQ)
- A Delay Register (DREG)
- An Instruction Cache (ICache)

The GPRs are available to store data which persists over multiple references, as well as software controlled environment variables (e.g. stack pointer, argument pointer, etc.). Each PE maintains its own set of GPRs, so a four processor configuration with a 32 element GPR set contains 128 registers (4 * 32).

Each PE also contains a FIFO queue (PEQ) for each PE in the system (including itself) to buffer data transfers between PEs. In addition, each PE has 2 memory queues (MQ's) which will buffer data requested from memory. While a single MQ would be sufficient (there is only 1 memory system), the frequency of generating two independent memory streams is great enough that an additional MQ has been allocated (as an architectural feature) to improved performance and simplify code scheduling. Providing two separate MQs allows two load requests to execute asynchronously (on different access PEs) and eliminates the need to read two data values off the same MQ during a single clock cycle [5].

The Output Queue (OutQ) buffers execution results that are scheduled to leave the PE (heading for memory and/or other PEs). This queue may seem unnecessary since there is no contention for the transmission bus (remember, each PE has a dedicated write bus); however, its use simplifies the issue logic of the PE by avoiding a pipeline stall at the end of the execution phase due to a *busy* line assertion. The operation of the Output Queue (OutQ) is not visible to the compiler.

In addition to the general purpose registers, there are two additional architecturally visible special purpose registers. The *Program Counter*, used to schedule the instruction flow, is automatically incremented after each instruction, and can also be modified explicitly by the execution of a branch instruction [6]. The *vector register*, *VREG*, is used to store the iteration count for vector instructions. Both the *PC* and the *VREG* are available as source inputs to all instructions in the same manner as the GPRs and the various queue elements.

The *Delay Register* is used to determine the number of *delay slots* that will be unconditionally executed following a branch instruction, and works as follows: A value specified by the first operand field in a branch instruction is placed in the delay register. The value in the delay register is then examined to determine how many instructions should be fetched after the branch instruction; as long as the value in the delay register is not zero, instruction fetch continues sequentially and the value in the delay register is decremented for each instruction executed. Once the delay register value reaches zero, the *PC* is updated to contain the branch target address (i.e. the branch is taken) [7]. This approach is a further generalization of the approach used in the PIPE processor, which is itself a generalization of the single delay slot employed in the RISC I processor [32].

Finally, each PE has an instruction cache to hold recently executed instructions and to relieve the UCache from the need to process instruction fetch from four separate processing elements.

3.5 Instruction Format

All instructions in MISC are 32 bits in length, and consists of an 8-bit opcode and four 6-bit operand fields as shown in Figure 3.3. The opcode field is separated into an instruction type field and a 6-bit opcode specifier. The operand fields may contain up to three source operand specifiers and a destination specifier.

[5] The size of each queue (PEQ and MQ) is an architecturally visible component; the compiler must know the size of each queue in order to schedule code correctly and avoid deadlocks due to resource depletion.

[6] This *PC* operates the same as the program counter in a scalar architecture.

[7] Conditional branches operate in a similar manner, but do not jump to the branch target if the condition is false

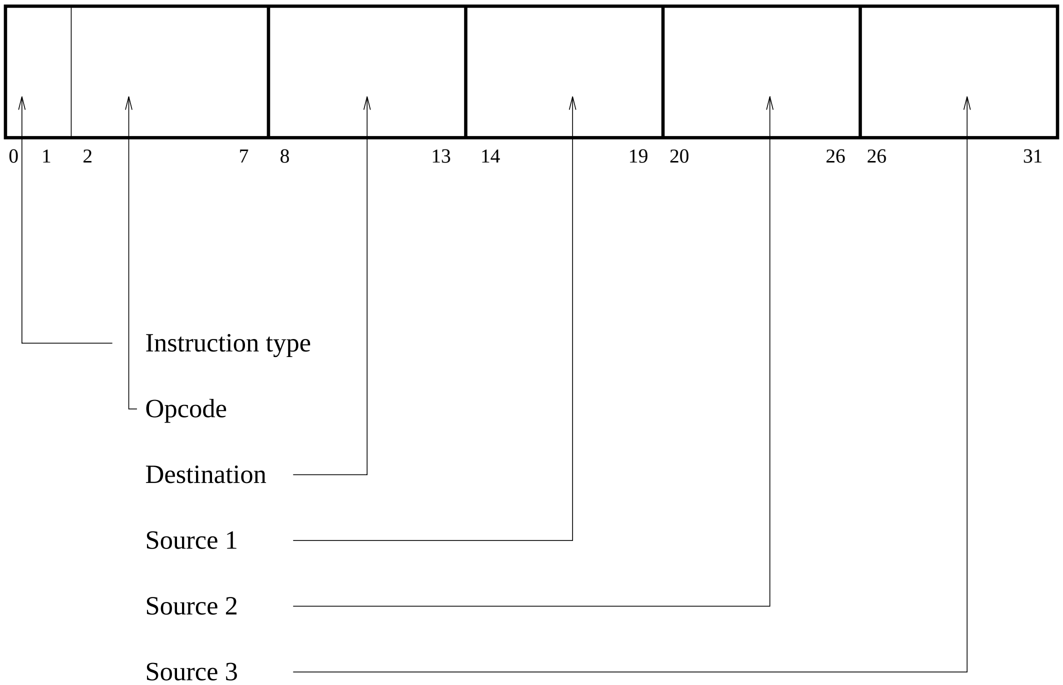

Figure 3.3: MISC Instruction Format

With 32 GPRs, only 5 bits are required to uniquely identify each general purpose register. Therefore, as shown in Figure 3.4, if the first bit is a 0, then the other 5 bits are used to address one of these 32 GPRs.

If the first bit is one, then the field specifiers are treated differently. In the case of a destination specifier (reference Figure 3.5), the remaining 5 bits are used as *routing* information for a data transfer onto the PE's PBUS [8].

For a source specifier, each operand can reference one of the input queues in a destructive or non-destructive manner. If the DQ bit is set, then the item read from the queue is de-queued; if the DQ bit is not set, then that item remains at the head of the queue. The two special registers (*PC* and *VREG*) can also be read in this manner.

At the instruction issue stage, if a queue is specified as a source input and that queue is currently empty, instruction issue ceases until all required input operands are available. This approach to handling empty queues simplifies the control logic of each processor; since each PE is a scalar processor, which does not support out-of-order execution, stopping instruction issue on an empty queue does not limit the ability to exploit any additional parallelism by the rest of the PEs.

[8]In a four PE configuration, 5 bits are sufficient to directly specify the route — one bit for each PE and one for the UCache. For PE configurations incorporating more than four PEs the 5-bit specifier is used as an index into a table of route bit-fields. The simulator places an artificial limit of 64 on the number of bits supported for each route specifier in the table; this means up to 63 PEs can be supported.

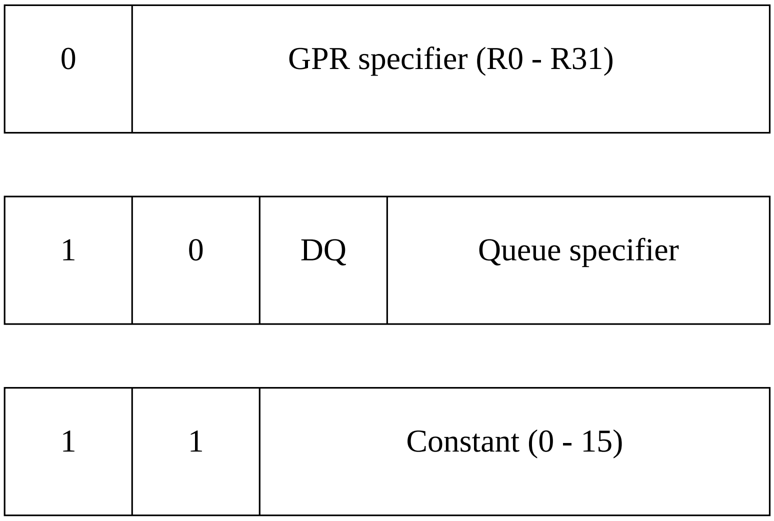

Figure 3.4: MISC Source Register Specifier Format

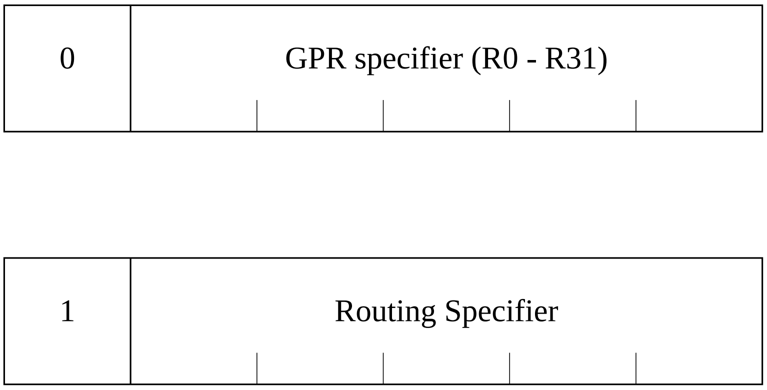

Figure 3.5: MISC Destination Register Specifier Format

Each source operand can also specify a small constant value. The ability to support a small constant for each operand in a three operand instruction provides considerable flexibility in code scheduling. Approximately 50% of all references to constant values generated by the compiler can be placed in this constant field [20].

Instruction Types

There are four different types of instructions shown in Figure 3.6: Scalar, vector, sentinel and predicate operations. [9]

Scalar instructions include ALU/FPU operations, load and store request instructions, control flow operations and some special purpose instructions used to control the execution model.

[9] A description of each instruction in the MISC design is given in Appendix A.

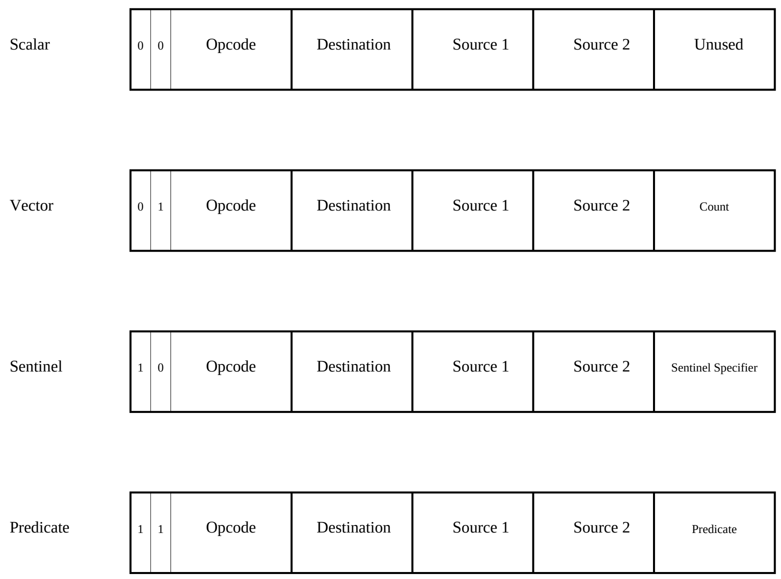

Figure 3.6: MISC Instruction Type

For example, invoking a memory read operation involves providing the UCache with the memory address of the data to be read, the set of destination PEs which are to receive the data, and the CBUS (and therefore MQ) on which the data should be placed. In a load request (LAQ) instruction, for instance, the *dest* field contains the set of PEs that are to receive the data. The address is calculated as a sum of the *src1* and *src2* operands and the destination MQ is specified in the opcode field; the LAQ instruction causes the MQ of the destination PEs to receive the data, while the LAQ2 instruction specifies the MQ2 queue as the location in which the data should be placed. The store request (SAQ) instruction operates in a similar manner, except that the *dest* operand specifies a single PE from which the UCache should receive data to be written to memory.

Scalar branch instructions are also fairly conventional. The *dest* field is used as a constant that states the number of *delayed branch* slots which contain instructions to be executed prior to starting instruction fetch at the branch target address. The address of the branch target is calculated as the sum of the *src1* and *src2* operands. Each of the other instruction types — vector, sentinel and predicate — is less conventional and will be discussed in greater detail in the next three sections.

3.6 Vector Instructions

MISC includes a class of vector instructions which perform certain scalar functions a specified number of times. There are three classes of vector operations: ALU/FPU, LAQ/SAQ, and a vector loop. Vector operations require an additional source operand specifier (*Count*), which is used as a vector count. When a vector operation arrives at the issue stage of the instruction pipeline, the vector register (*VREG*) is set to zero and the vector count register (*VCOUNT*) is loaded from *Count*. A single execution of the instruction is then performed and the *VREG* is incremented [10]; this repeats while the value in *VREG* is not equal to *VCOUNT*. Once the value in *VREG* equals *VCOUNT* normal instruction sequencing continues. The semantics of this mechanism are:

VCOUNT = Count;
for (VREG = 0; VREG < VCOUNT ; VREG++) dest = src1 op src2;

3.6.1 Vector Loop

The use of the vector register in conjunction with the delay register provide a means of extending the use of vector instructions to iterate on sequences of instructions (whereas the vector ALU operations iterate only a single instruction). The use of the vector loop instruction can also eliminate the need for branch instructions to control the iteration of simple (single basic block) loops. The vector loop, VLOOP, instruction can be used when the number of iterations that a loop should execute is known at the initial entry into the loop (either as a constant or register variable). When a VLOOP instruction is issued the following activities occur in parallel:

- load the *DCOUNT* with the delay count field of the instruction (this is the same as conventional MISC branch instructions)
- set the value of *DREG* and *VREG* to zero
- copy the value specified by the *Count* field of the instruction to the *VCOUNT* register
- set the jump address (*JUMPPC*) to the first delayed instruction (i.e. *PC* + 1)

At this point instruction flow continues as in a branch instruction; instructions are executed from the delay slots until *DCOUNT* instructions have been executed. Once all the instructions in the delay slots have been executed the *VREG* is incremented and compared to *VCOUNT*. If *VCOUNT* iterations have not yet been completed (i.e. *VREG != VCOUNT*) then the *JUMPPC* is placed in *PC*. This initiates the next iteration of the loop. When all iterations have been executed (*VREG = VCOUNT*) execution continues with the instructions following the loop.

[10] The VREG is incremented up to VCOUNT. An alternate approach would be to start VREG equal to VCOUNT and decrement until it reached zero. The first approach was used because it simplified the application of compiler transformations dealing with induction variables (as described in chapter 4).

3.7 Sentinel Instructions

While vectors of specified length are the most common organization for variable length arrays, a significant alternate mechanism is the use of a trailing sentinel (special) value to mark the termination of a string of data. This mechanism is of particular importance because it is the method used by the C libraries to manipulate character string data. MISC can perform efficient string manipulation through its class of *sentinel* instructions. These instructions use the *Sentinel Specifier* field of the instruction to reference a register that is compared to the sentinel value (assumed to be zero in the initial design). If the value in the register identified by the *Sentinel Specifier* does not equal the sentinel (zero), then the execution of the scalar version of the sentinel instruction is allowed to issue. The semantics of this mechanism are:

while (R[Sentinel Specifier] $\neq$ *0) dest = src1 op src2;*

Conventional wisdom holds that implementing higher level semantics at the instruction level seldom leads to performance improvements because of the complexity of implementation and the scarcity of application. We believe that the vector and sentinel instructions defined here can be implemented with minimal hardware modification to the issue logic. Furthermore, the MISC approach of multiple instruction streams leads to a greater potential for application of these instructions; often an application of the construct (a *while* loop in this instance) cannot be made because of the complexity of the test condition in the application for a normal machine. In a single instruction stream design, a single processor must evaluate the test condition and proceed with the loop or exit accordingly. The requirement to both evaluate an arbitrarily complex test condition and perform the control flow operation cannot be efficiently reduced to a single instruction. However, in MISC one PE can evaluate the complex test condition and broadcast a boolean result to all other PEs, which are left with simple boolean tests.

3.7.1 Sentinel Loop

Much like in the vector loop instruction, the delay register can be combined with the sentinel mechanism to provide another simple branch hiding instruction. Sentinel loops share much of the control logic that is used in vector loops; the manipulation of the *PC*, the use of the delay slots and the termination of the loop are essentially the same. These two loop instructions differ only in the way the exit condition is calculated; a simple register comparison is performed for sentinel loops while a counter is incremented and tested for vector loops. When a sentinel loop, SLOOP, instruction is issued the following activities occur in parallel:

- load the $DCOUNT$ with the delay count field of the instruction
- set $DREG = 0$

- save the sentinel register specifier (from *Sentinel Specifier* of the instruction)
- set the jump address (*JUMPPC*) to the first delayed instruction (i.e. $PC + 1$)

At this point instruction flow continues as in a branch instruction; instructions are executed from the delay slots until $DCOUNT$ instructions have been executed. Once all instructions in the delay slots are executed the *sentinel* is retrieved and compared to the terminating value (0). If the value is not equal to 0 then the *JUMPPC* is placed in *PC*, and the next iteration of the loop then proceeds. Once the sentinel value equals 0, instruction fetch continues sequentially from the end of the delayed branch instructions exiting the sentinel loop.

3.8 Predicate Instructions

Predicate instructions conditionally execute code by controlling the writeback stage of the pipeline with a conditional value. If the condition is true (non-zero for MISC) then the writeback stage is allowed to modify the state of the machine; if the condition is false then the writeback is prohibited. Instructions of this type allow the removal of numerous branch instructions by replacing the conditional instruction issue (through the use of branches) with a conditional writeback (through the use of predicates); the results are the same, but a control flow operation can be eliminated and the ability of the compiler to aggressively schedule code blocks can be improved.

In the MISC architecture the *Predicate* operand is used to specify the condition value. The semantics of the predicate operation is:

$dest = src1$ op $src2$ if $Predicate \neq 0$

The transformation of a conditionally executed code block to a predicate instruction can be seen in the following fragement of C code:

if (Predicate) dest = src1 + src2

Without predicate instructions, that code would translate into the following assembly code sequence:

```
            branch_zero     Predicate,  Label1
            add             dest,  src1,  src2
Label1:
```

With predicate transformation the branch can be removed and a predication form of the addition can take place, leaving the following assembly language instruction:

```
            add_predicate   dest,  src1,  src2, Predicate
```

Conditional branches themselves predicate instructions which operate in the same manner as the scalar (unconditional) branches except that the *Predicate* operand specifies the register to be tested to determine whether the branch will be taken or not.

3.9 Data Cache Structure

The Data Cache unit consists of one or more single access data caches of unspecified size, a PBUS request and data interface, a cache manager, a memory interface, and a CBUS manager. The design goals for this component of MISC are somewhat different than those for on-chip data caches for other processors. While on-chip caches are generally used to reduce memory access latency, this function is less vital in a decoupled machine designed specifically to tolerate long memory latencies. However, with four processors on the same die, it becomes imperative to reduce the off chip memory activity to avoid over-taxing the I/O capabilities of the chip. A majority of this reduction is performed by the instruction caches located on each processor; it is the responsibility of the data cache to further reduce off chip traffic to a minimal level.

Memory operations are separated into two components: A memory request is initiate by an *access* processor and the data is either generated or used by some other processor. When a memory load operation is initiated, for example, the *access* PE sends the address of the desired memory location and a set of destination PEs to the UCache; the value at that memory location is then retrieved from the cache, in the event of a cache hit, or memory, for a cache miss, and returned to the PE or PEs specified by the request.

When a memory store operation is initiated two events occur in an unspecified order; an access PE sends the address of the storage location to the cache and another PE (possibly the same PE) transmits the data to the cache over its internal bus; these events can occur in any order. In either case, the address and data will eventually be placed into buffers and the cache operation will proceed when both the address and the data have arrived.

3.9.1 Design Goals

Much of the hardware complexity found in the MISC design is located in the memory interface. This is because the memory system is inherently a shared (global) resource, so it is responsible for managing the requests from asynchronous sources (PEs) and interleaving them in a consistent and efficient manner. Furthermore, as more instruction level parallelism is found, the memory system (and the cache) must be able to support multiple operations each cycle. The MISC cache organization is designed to maximize the ability to handle multiple, independent references.

The goals of the MISC memory systems include:

- The ability to handle both individual and group broadcast memory requests.
- Allowing multiple memory requests to be satisfied in each clock cycle from one or more PEs.
- Maximizing memory request throughput.

- Combining inter-cache-line memory requests to improve cache performance.

3.9.2 Component Structure

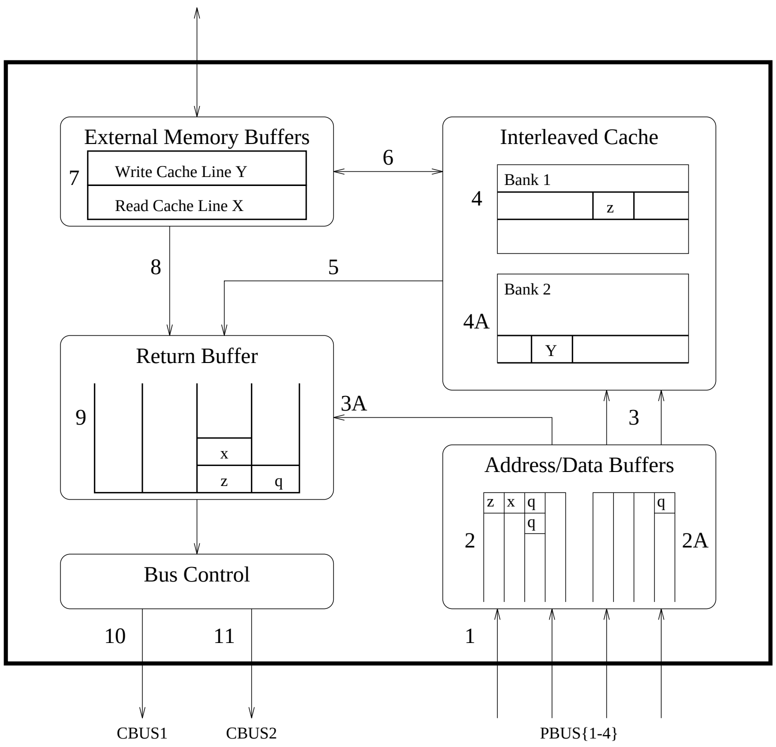

Figure 3.7: MISC Cache: Component Design

The memory system and cache organization is separated into five components shown in Figure 3.7. These components are:

- Address/data buffers: These buffers receive the effective address for load instructions and both effective address and corresponding data for memory store instructions.

- Interleaved cache memory: A cache memory consisting of a number of single access caches (e.g. four) using bits in the effective address to determine the appropriate cache bank [11].

- External memory buffer: An interface between the cache and the external memory system. The external memory buffer maintains a list of outstanding

[11] An interleaved cache is different than a multi-ported cache, in which any four requests can be performed from arbitrary locations. An interleaved approach trades a reduction in performance due to a more restrictive access mechanism (in which references to the same bank must be serialized) for a significant simplification in implementation.

cache replacement requests, including those current cache lines marked for replacement which have been modified as well as the effective addresses of requested cache lines.

- Return buffer: Buffers load data destined for transfer to the processing elements. The return buffer is also responsible for reinstating the original order of memory requests.

- Bus control: Controls access to the internal buses used to transfer data from the cache unit to the processing elements.

To help explain the function of the cache, the following example, consisting of a simple expression involving four memory references, will be used:

$q = q + z[i] + x[i];$ [12]

This example, shown in Table 3.1, will be referenced throughout the rest of this section to demonstrate the operation of the UCache.

Table 3.1: An Example Expression for Memory References

Operation	Dest PE	In Cache?	Effective Address (32 bits)	Cache Tag (18 bits)	Bank Selector (1 bit)	Line Selector (7 bits)	Line Offset (6 bits)
PE3: laq	PE4	Hit	q = 8878DH	22H	0	3EH	0DH
PE1: laq	PE3	Hit	z[] = B48A2H	2DH	0	22H	22H
PE2: laq2	PE3	Miss	x[] = 4FFC4H	13H	1	7FH	04H
PE3: saq	PE4	Hit	q = 8878DH	22H	0	3EH	0DH

In this example, the memory operations performed are the loads from each array (*z[]* and *x[]*) and a load followed by a store of variable q. The memory references are shown in Table 3.1. The table columns show:

- Operation: the operation performed to initiate the memory request (in the form: "*access* PE: instruction type").

- DEST PE: the PE which will receive the data from the cache unit.

- Effective Address: the variable name and its effective address.

- In Cache?: whether or not the reference is in the cache.

[12] The four references are *q,q,x[],z[]*; the variable i resides in a register in this example.

- Cache Tag – Bank Selector – Line Selector – Line Offset:

  The effective address is broken down into four components for further processing by the cache. The Cache Tag is the most significant 18 bits of the (32 bit) effective address and is used to allow a bi-directional mapping of a cache line to a memory location. The Bank selector is the next most significant bit and is used to select the cache bank to search for the reference. The line selector determines which cache line is referenced; a 128 line, direct-mapped cache configuration (requiring a 7-bit line specifier) is used. Finally, the line offset (6 bits for a 64-word cache line) specifies which entry in the line contains the value at that effective address.

3.9.3 Initiating a Memory Operation

When a memory reference is made (either a load or a store) a processing element (the access PE) initiates the request in the following manner:

- The effective address of the memory request is placed on the data lines of the internal bus associated with the access PE.
- The memory operation control line is set to indicate that this communication (through the PE internal bus) is meant for the cache unit. This instructs all PEs to ignore the transfer and the cache unit to process the request.
- The route lines of the access PE are set to identify the source or destination of the memory operation [13].
- The R/W control line is set or cleared to specify whether the request is for a load or a store operation.
- A value is placed on the ReturnQueue control line to specify which memory input queue is to receive the requested data item.

A data STORE operation is performed by specifying the UCache in the DEST operand field; the PE generating the data will place it on the internal data bus, specifying the cache unit as the destination and the UCache then enqueues the data into the data buffer associated with that PE. Storing data to the memory system is initiated by the access PE executing an SAQ instructions. This first step is labeled with a **1** near the PBUS inputs to the cache in Figure 3.7.

[13] If the requested operation is a memory STORE, the route lines specify which PE will supply the data to be stored. If the requested operation is a memory LOAD, the route lines specify which PEs should receive the value read from memory. This information will later be used by the cache unit to set the route lines for the return of the value.

In the example, there are three LAQ and one SAQ instructions. These instructions can be executed by one, two or three independent PEs. [14]

3.9.4 Address/Data Buffers

Once a memory request is received by the cache, the effective address is placed in the address buffer associated with the access PE. If data is sent to the cache, that data will be placed in the data buffer associated with the PE sending the data. It is the responsibility of the address and data buffers to store this information in preparation for the cache read or write operation. The address buffer will also reorder memory operations (if possible) to improve cache performance; this may also require reordering the contents of the data buffer to maintain consistency between the two buffers. The structure of the address and data buffers are shown in Figure 3.8.

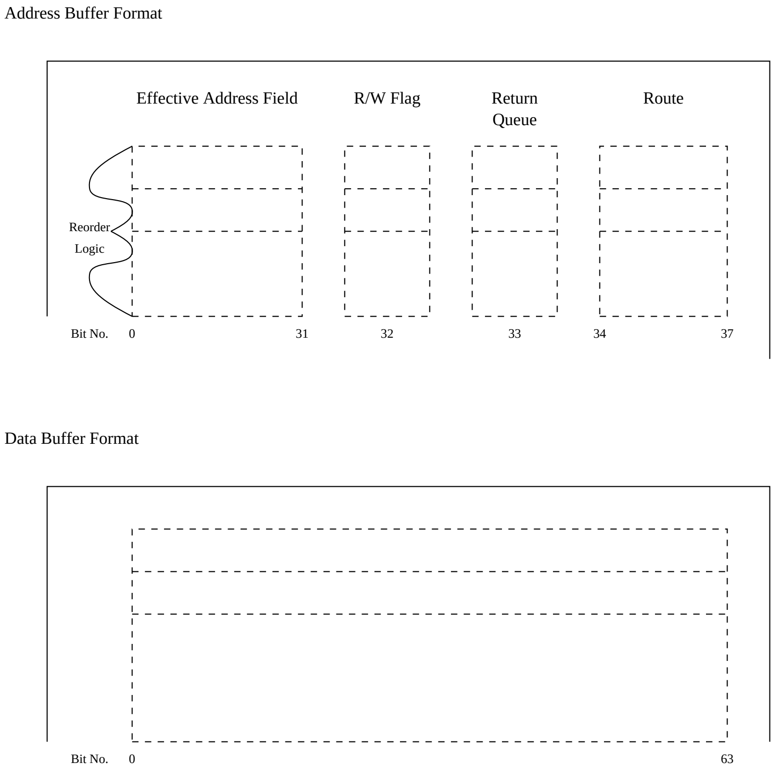

Figure 3.8: MISC Cache: Address/Data Buffer

Each entry in the *address* buffer has four fields: effective address, read/write flag, ReturnQueue and route specifier. Each of these fields contains information

[14] References to variable q (both the load and the store) must be initiated on the same access PE to preserve correct ordering.

needed by load operations, while store operations do not use the ReturnQueue field. Each entry in the *data* buffer has only a single field containing the data value to be written to the cache. This information will reside in the data buffer until paired with a store request for further processing by the interleaved cache unit. The processing of requests depends on the request type (R/W). The write state indicates that a memory store operation should be initiated; a read state initiates a memory load operation. The next two sections will describe the operation of memory load and store instructions in more detail.

MEMORY LOAD OPERATION

Load requests are placed in the address buffer of the access PE; since no data is associated with the load request there is no need for corresponding entries in the data buffer. Processing requests in the address buffer involves the selection of the entry to forward to the cache memory; in a simple implementation, requests can be processed in the order in which they are received. However, in order to increase the overall cache throughput, it is often useful to reorder requests. Therefore, on each cycle the first N entries in the address buffer are scanned for load requests that are ready to be forwarded to the cache cell array. The requests which are ready to proceed will then be forwarded to the cache.

As a load request is sent to the cache, the destination route is simultaneously sent to the return buffer. This will eventually match up with the data value read from the cache unit; even if a cache miss occurs, the data will eventually make its way to the cache and through the cache to the return buffer.

The example code includes three load requests which are placed into the address buffers of PE1 through PE3 (Figure 3.7 step 2). The array elements (z and x) are destined for PE3 and the variable q is destined for PE4. When the load requests are ready to be serviced by the cache, two events occur for each request. First, the effective address of the load operation is sent to the interleaved cache unit (Figure 3.7 step 3). At the same time an entry in the return buffer is allocated to store the data once the cache has obtained the correct value (either directly or after a cache replacement - reference Figure 3.7 step 3A).

MEMORY STORE OPERATION

As discussed earlier, a store request consists of an address specified in the SAQ instruction and a specification of the PE which will supply the data to be stored. The store request will stay in the address buffer of the access PE until conditions allow it to be forwarded to the interleaved cache unit. The route specifier of the store request is used to match address buffer entries to data buffer entries. Once the data is available, the store may proceed. If the data arrives first, it sits in the data buffer until matched to a memory store request. It is the responsibility of the compiler to schedule instructions such that no ambiguity can exist in the address/data matching process.

The store operation in the example will be initiated when the SAQ request arrives from the access processor. The effective address (q) is specified, but now the route specifier identifies which data queue will contain the data to be stored in the cache (PE4). The store request will remain in the address buffer until data is sent to the appropriate data buffer. Once that data arrives (Figure 3.7 step 2A), the store request can be forwarded to the cache unit. It is also the responsibility of the compiler to ensure that no race conditions exist with multiple store requests targeting the same data queue.

3.9.5 Interleaved Cache Memory

When all the components for a memory request arrive at the UCache, the request can be forwarded to the interleaved cache. Once a cache line is selected, the tag bits are compared to see if the current line in the cache matches the reference address. If the tag bits match the request can be processed. A store will modify the contents of the cache location, and the cache line will be marked *dirty* to signify that the contents differ from those found in the memory system; when this line is deallocated, the line must be written to memory. A load will place the line into a multi-ported line buffer and allow the line offset (or multiple offsets if combining has occurred) to fetch the desired data and forward it to the Return Buffer [15]. The processing of requests in the cache unit proceeds in a standard manner.

If the reference tag does not match that of the cache line tag a cache miss occurs. Cache miss processing requires a line replacement request be forwarded to the External Memory Buffer. This request contains the address of the original cache request so that the data can be retrieved from memory and placed in the cache. If the current cache line is dirty, then an additional external memory operation will be initiated to write the modified cache line (the one that will be replaced) out to memory. The cache then continues processing further requests until the data is available.

Referring again to the example, assume that the *z[]* and q requests are present in the first bank of the cache and the *x[]* request is absent (Figure 3.7 step 3). In this case, the data associated with requests *z[]* and q will be sent to the return buffer (Figure 3.7 step 4). Since the *x[]* request requires a cache line fetch from memory, this modified request will be forwarded to the external memory request buffer (Figure 3.7 step 5). In addition, if the replaced line (line y in the example) is dirty, a cache line store request will also be forwarded to the external memory request buffer.

[15]In the MISC design this is implemented as an N-way (where N is a power of 2) interleaved cache, splitting cache accesses across N independent cache banks. Two references can conflict when the bank selectors for the references match.

3.9.6 External Memory Buffer

The external memory buffer accepts requests from the cache memory and provides the interface between external memory and the interleaved cache. External requests are for complete cache line loads and stores. Requests are enqueued in the buffer, but may be reordered to allow loads of non-dirty lines to precede earlier requests.

The example stream requires two external memory operations: the store of the dirty line (y) and the load of the cache line associated with request *x[]* (Figure 3.7 step 6). To achieve optimal performance, the load request for the replacement line should be processed before the contents of any dirty lines are written back to memory. This requires an additional line buffer (associated with the external memory buffer) to hold the data from the line(s) being replaced. Once this data is copied from the cache to the external memory buffer, the load may precede. On completion of the cache line fetch, the data will be placed into the cache line and then forwarded to the return buffer. The store of the dirty line can then continue without affecting the latency of the original load request (Figure 3.7 step 7).

3.9.7 Return Buffer

Memory references must be returned to the processing elements in program order due to the FIFO semantics of the load queues (MQ1 and MQ2). Reinstating the original order of references is the function of the return buffer.

Once a memory request has been sent to the interleaved cache an entry in the return buffer must be allocated. Requests enter this buffer in the same order as the originating load requests entered the address buffer. When the desired data is retrieved from the cache (this could be much later in the event of a cache miss) it is forwarded to the return buffer and marked as ready to return to the PE(s) via the control bus. The data will not be sent to the PE(s) until all prior load values destined for the same PE(s) have been sent. Figure 3.9 shows the structure of the return buffer.

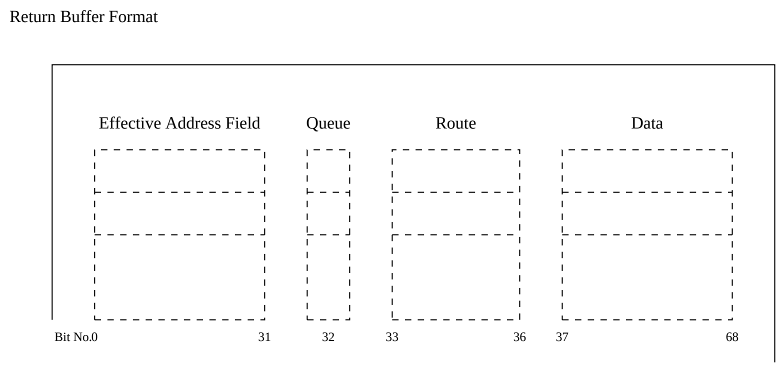

Figure 3.9: MISC Cache: Return Buffer Design

Each entry in the return buffer contains:

- the route map specifying the destination PEs. This will be used to set the route lines of the CBUS when the data is sent to the destination PE(s).
- a memory queue specifier used to determine which CBUS (and therefore which PE memory queue) will receive the result of a load operation.
- the effective address of the request — needed to determine which data item corresponds to each entry. When the data is retrieved from the cache bank, the effective address is used to identify the entry in the return buffer.
- data field for one (32-bit) word of memory. This is required since a value may reside in the return buffer for an arbitrary amount of time depending on the reordering requirements of previous load operations.

A complete reinstatement of the original order is not always necessary; when the intersection of the route map is empty (i.e. the destinations of two load requests do not target the same PE) or the memory queue specifier is different, the requests can precede in any order (relative to each other).

When the load requests in the example were forwarded to the cache unit, information was also sent to the return buffer to reserve an entry (Figure 3.7, step 3A). Once processing by the cache unit completed for references q and *z[]*, the data was forwarded to the return buffer (Figure 3.7, step 5). The processing of reference *x[]* was delayed until the data could be retrieved from the external memory system; upon its arrival, the data was forwarded to the return buffer (Figure 3.7, step 8). In this example there is no re-ordering that must occur since no conflicts reside in the destination specified between the three references; reference q is destined for PE 4 while reference *z[]* and *x[]* are assigned different memory queues (in PE 3), so no ordering relationship has to be maintained. This allows the data items to be sent to the PEs as soon as they arrive, assuming that the destination memory queues are not full before the processing of this iteration of the loop (Figure 3.7, step 9).

3.9.8 Bus Control

The bus control unit is responsible for transferring values from the return buffer to the destination PEs. CBUS access for the cache is performed in the same manner as PBUS access for each PE; when all destinations are ready to accept the transfer, the data and routing information is placed on the bus. The correct return bus (CBUS1 or CBUS2) is determined by the bus specifier (originally specified by the load request instruction). This action completes the processing of the load request.

The example includes three operations involving the cache Bus Control unit. The load of reference q and *x[]* returns data on CBUS1 (Figure 3.7, step 10) while the data for *z[]* is returned on CBUS2 (Figure 3.7, step 11). Busy lines are tested first, and when the memory queues are available the route specifier and data is placed on the bus completing the transaction.

3.10 Summary

This chapter described the architecture of the MISC processor. The MISC design incorporates some unique features in order to achieve high performance processing. These include:

- A separation of functional units into independent processing elements capable of very close cooperation to help extract available parallelism.
- A more general approach to register addressing than previously found in superscalar or decoupled architectures, which allows an arbitrary number of registers to be scheduled by the compiler without requiring additional operand specifier bits.
- Flexible routing control to enable instruction level broadcast operations to transmit data between PEs.
- A unified cache design which increases data throughput while supporting multiple processing elements and a broadcast capability.

The design and interaction of these features was described, including a description of a complete instruction set architecture capable of specifying the interaction of the processing elements through architecturally visible queues, the design of a high throughput data (and instruction) cache, and a series of special purpose vector, sentinel and predicate execution modes.

Chapter 4

Design of the MISC Compiler

One open question with a distributed architecture like MISC is how well the translation process from high level to machine code can be incorporated into a compiler. Is it possible to efficiently compile C programs into separate instruction streams? Are the architectural choices that made decoupled designs useful in executing scientific codes applicable to more general applications? These questions can only be answered by constructing a compiler and testing various code separation models.

Two aspects of the translation must be addressed by the compiler in order to obtain high performance execution:

1. *Achieving a balanced partition of instructions.*

2. *Minimizing the impact of memory latency.*

This chapter describes the structure and performance of the MISC compiler including analysis of several code partitioning strategies to determine how well the execution of the program instructions can be balanced across multiple processors. The scheduling of register queues is examined to quantify their affect in reducing the impact of memory latency.

4.1 The MISC Compiler Overview

The compiler and code scheduler for a high-performance architecture requires a high degree of sophistication in order to realize the full potential of the hardware. Independent instructions must be assigned to operational units in a manner that minimizes the number of cycles in which no instructions can be issued. The task of the scheduler in a multi-issue system is further complicated by the fact that while the latency of operational units and memory may remain fixed, the number of instructions that must be scheduled in a period is increased as the width of the issue stage increases.

Several studies [33] [34] [35] indicate that compilers using relatively simple scheduling techniques are capable of identifying 2-3 independent instructions per cycle. Other studies [36] [37] [38] suggest that even more parallelism can be found if the compiler's scheduler is capable of performing extensive code motion and more sophisticated global scheduling, or if the hardware is capable of reordering the original schedule and speculatively executing instructions around basic block boundaries.

The MISC system was designed to allow out-of-order execution without the need for complex reordering hardware. The MISC compiler is responsible for identifying instruction sequences which have no dependencies, or in which the dependencies that exist can be scheduled across PEs without requiring circular dependencies within groups of PEs. These code sequences are then scheduled on different PEs. Each of these instruction sequences may containing code that contains tightly interconnected dependencies; these interrelated instructions are placed on the same PE to allow the general purpose register file to serve as the primary means to communicate dependencies. The *very portable C compiler* (vpcc) [39] under development at the University of Virginia served as the base compiler for MISC [1].

The first step in compiling an application is to translate the C code into an intermediate description. The MISC compiler uses a *Register Transfer List* (or RTL) form for the intermediate representation; this form is similar to that used by the *gcc* compiler. This transformation is performed by the vpcc front end. Once an application has been translated into RTL form many standard transformations can be performed. The code generator then translates the RTL description of a program into parallel machine code for the MISC machine. An overview of the optimization algorithm is presented in Figure 4.1.

During the optimization phase of the compilation process, a number of code transformations are applied, including many conventional transformations that do not require information specific to a particular architecture. These transformations include common sub-expression elimination, dead code removal, strength reduction, and many others [30]. It is best to do these transformations at this point, before the complexity of inter-PE dependencies must be considered. Similarly, *IF-conversion* [40] can also be performed at this point in order to simplify the control flow. Global dataflow analysis can also be performed, and a *Program Dependence Graph* (PDG) constructed.

Once the conventional optimization have been applied, the MISC compiler begins the process of partitioning instructions across the multiple processing elements of the MISC architecture. The code partitioning phase separates the operations required by the program into multiple (virtual) processing elements in a manner that maximizes the number of processing elements utilized. The processor load balancing phase of the compiler then re-partitions the schedule to evenly distribute

[1] Other compilers were explored for this dissertation, but lacked the flexibility and/or dependability of vpcc.

```
foreach function
        load initial RTL description provided by front end
        conventional (single stream) transformations
                initial register allocation
                IF-conversion
        build program dependence graph

        partition code
        balance code

        foreach PE
                branch reduction
                secondary register allocation
                schedule code
        output MISC machine code for function
```

Figure 4.1: Structure of the Compiler

the operations onto the number of physical processing elements available on the target machine. Once the instructions have all been allocated to the PEs, more code optimizations can be applied to each of the PE instruction streams. Many of the standard transformations described previously can be applied again to further improve performance in each individual stream (with new restrictions to maintain inter-PE dependencies) [2]. Finally, each instruction stream is scheduled and the MISC machine code is generated.

The code scheduling method used by the MISC compiler exploits the asynchronous behavior of the processing elements to provide many of the characteristics found in software pipelining [41]. Software pipelining is a compiler transformation technique (first developed for VLIW architectures) that can be applied to simple loops to eliminate the effects of high latency operations. Software pipelining can be viewed as an improved form of loop unrolling, where multiple iterations of the loop can be scheduled as if they were part of a single basic block. This allows instructions from different iterations of the same loop to be integrated into an optimal schedule. Similarly, in MISC, individual PEs can be executing instructions originating from different iterations, while the PE queues perform a simplified form of register renaming.

The SPEC92 [42] integer benchmark applications will be used throughout this

[2]Re-application of standard transformations (e.g. copy propagation) after code partitioning can occasionally locate code sequences that were obscured by intervening instructions before partitioning.

chapter to measure the performance of code generated by the compiler after the various stages of optimization/translation [3]. These applications are described in Table 4.1.

In addition to the SPEC92 integer benchmarks, four additional applications will be evaluated that have program characteristics more amenable to extracting ILP. These include one application, *ear*, from the SPEC89 [43] floating point benchmark set; this is the only application in the SPEC89 or SPEC92 floating point suite that is written in C (the rest are written in Fortran). Two other benchmark programs are applications common to other areas of computer science: *povray* [44] is a popular computer graphics imaging program and *sobel* [45] is a program which implements a widely used convolution filter to enhance a bitmap image in the computer vision field. The final benchmark, *KMP* [46], is the classic sub-string matching algorithm developed by Knuth, Morris and Pratt.

To help illustrate each phase of the code generation process, two simple examples will be used. The InnerProduct() function is taken from Livermore Loop 3 [47] and is a highly parallelizable program, while the LinkedList() example demonstrates how loops with more complicated control flow can be partitioned. The code for these examples is shown in Figures 4.2 and 4.3. The following sections examine the operation of the compilation phases in more detail.

[3] These include those benchmarks in the SPEC92 suite written in the C language that have shown the greatest resistance to exploiting instruction level parallelism.

Table 4.1: Benchmark Application Descriptions

Benchmark	Description
compress	A file compression program, version 4.0, that uses adaptive Lempel-Ziv coding.
eqntott	A translator from logic formula to a truth table.
espresso	A logic optimization program, version 2.3, that minimizes boolean functions.
gcc	A benchmark version of the GNU C Compiler, version 1.35.
xlisp	A lisp interpreter that is an adaptation of XLISP 1.6.
sc	A spreadsheet program, version 6.1.
povray	A graphics ray tracing program, version 3.0.
sobel	A computer vision convolution filter used to highlight the edges of a bitmap image, version 1.3.
ear	EAR simulates the propagation of sound in the human cochlea's (inner ear) and computes a picture of sound called a cochleagram, version 1.1.
KMP	A linear-time string-matching algorithm developed by Knuth, Morris and Pratt which locates substrings in a text array taken from [48]

4.2 Structure of the MISC Optimizer

The MISC optimizer uses existing techniques and conventional transformations when possible; for those optimization that are unique to MISC, or where existing techniques require modification (e.g. register allocation incorporating queues), care has been taken to maintain the same level of complexity found in most current optimizers.

```
int inner_product() {
int k, q=0;
    for (k=0; k<1024 ; k++)
        q = q + z[k] * x[k];
}
```

Figure 4.2: Example Code: InnerProduct()

```
int linked_list_example() {
int *list, value=0;
    for (list=head; list ; list = list->next)
        if (list->x != 0)
             value = value + list->z / list->x;
}
```

Figure 4.3: Example Code: LinkedList()

4.3 Conventional Transformations

Many conventional optimizations can be applied to the intermediate code before applying the algorithms specifically designed for MISC. Two standard optimizations that warrant special mention in their relationship to the MISC approach are *register allocation* and *IF-conversion.*

4.3.1 Register Allocation

Register allocation for a decoupled machine requires additional analysis by the compiler because the extensive use of queues requires an ordering constraint on register use that differs from conventional register allocation.

The vpcc compiler supports the specification of these ordering constraints in the allocator; this greatly simplifies the MISC translation process. Standard register allocation methods can be used, with one exception: MISC has a large number of *register classes*, unlike most architectures (which have only 2 register classes, integer and floating point). Since each PE has a general purpose register class, two separate memory input queues, and a complete interconnection of inter-PE transfer queues, a four processor MISC machine has 28 different register classes (1 GP x 4 + 2 MQ x 4 + 4 PEQ x 4). In addition to the large number of register classes, all but the general purpose registers are FIFO queues. If the allocation of a new register instance would violate the FIFO ordering of the queue, for example, the allocation is disallowed and the architectural register dependency remains.

In order to provide a compact two byte representation of any machine register, the intermediate RTL format employed by the vpcc compiler reserves 4 bits to identify one of 16 different classes and 12 bits to identify the register within that class. Since this format is incapable of accurately representing a full MISC machine, the MISC register class model must be modified [4]. The modified model supports only unidirectional communication between PEs through the PE transfer queues — PE1 can send data though queues to PE2-PE4, PE2 can send data the PE3 and PE4, and PE3 can send data to PE4. The queues to send data back cannot be represented in the existing RTL format. Fortunately, the code transformation strategy employed in MISC rarely requires data to be transmitted back to PE1, and in those cases when it is necessary, a transfer through memory can be performed. Forcing uni-directional data flow through queues means that one half of the inter-PE queues will be unused. This may have a marginal benefit in the scalability of the system when the code executed is required to meet these partitioning constraints.

Function calls also pose an interesting problem within a tightly coupled MIMD architecture. When a function call is made, any variable may be passed as a parameter to the function. One standard compiler technique to improve the performance of function calls is to place the first few parameters in registers before executing the call. However, in MISC variables are distributed among the PEs. How should parameters be passed in MISC during a function call? We chose to place all parameters on the stack — while this does not provide the best performance, it works and simplifies a number of problems with incomplete dataflow analysis between functions. Furthermore, the highly localized reference patterns along with the use of the MISC UCache mitigate much of the potential performance degradation.

[4] It is unfortunate that the designers of vpcc chose to implement such a restrictive register representation. They chose this representation to reduce the RTL size and obtain a very minor improvement in parsing speed at the cost of generality. A better approach would be to encode the register class and register identifier as space delimited ASCII strings.

4.3.2 IF-Conversion

IF-conversion is an optimization technique that converts control flow operations (branches) into predicate operations in order to reduce the effect of control dependencies on program performance when possible. The MISC compiler uses the predicate instructions, described in section 3.8, to replace short forward branches; instead of conditionally issuing those instructions by preceding them with a conditional branch, the branch condition is calculated and placed in a register used to control the completion (writeback stage) of the instruction execution. This results in a reduction of the number of branches in the original code.

The advantage of IF-conversion is more pronounced in MISC than in other architectures because of the need for duplication of branch instructions across all PEs. In MISC, the IF-conversion algorithm removes not only the original branch instruction but all the duplicates as well. Table 4.2 shows the resultant reduction in branches due to the use of IF-conversion on the set of applications chosen for this study. The first column identifies the benchmark application, the second gives a dynamic count of the instructions executed, the third column gives a dynamic count of the number of conditional branches, and the fourth specifies how many of those conditional branches can be eliminated using predication. This table shows that on average approximately 8 percent of conditional branches can be eliminated and replaced by short sequences of predicated code. In some cases, most notably *KMP*, a much greater number of branches can be removed. This will reduce the branch duplication problem and improve the performance of the code scheduler by increasing basic block size [49].

4.4 Dependence Graph

Following the initial conventional optimizations, a dependence graph is generated to support the code partitioning and load balancing phases of the compiler. To construct the graph, instructions are associated with nodes in the graph and true dependencies are associated with directed edges. Control dependencies are also represented as directed edges identifying a dependence between a branch instruction and each instruction in the basic blocks following the branch (both fall-through or branch target blocks). Other dependencies are added in order to construct a precedence ordering [50]. A precedence ordering augments the dependence graph with additional dependencies required to identify possible memory conflicts; these conflicts can occur when the compiler can not guarantee that two distinct memory operations do not map to the same memory location. In this case, the original order of the operations must be maintained by including additional dependencies in the precedence graph.

Table 4.2: Reduction of Dynamic Branch Executions Due to IF-Conversion

Program	Total Instructions (in thousands)	Conditional Branches (in thousands)	Removed Using If Conversion (in thousands)
compress	83,947	11,739	1,234
eqntott	1,395,165	342,595	1,020
espresso	521,130	76,466	5,306
gcc	142,359	21,579	4,132
xlisp	1,307,000	147,425	8,922
sc	889,057	150,381	28,120
povray	1,438,399	335,702	52,311
sobel	342,676	58,122	0
ear	17,010,166	1,311,243	71,025
KMP	932,541	182,031	74,924
InnerProduct	9	1	0
LinkedList	10	2	1

4.5 Code Separation

A grasp of the concept of a *leading* (or *lead*) processing element is central to the understanding of code separation. In a MIMD architecture, each of the instruction streams executes independently (ignoring for a moment any data dependencies). Therefore, if operations are scheduled carefully, some of the streams can be allowed to proceed farther ahead in the computation than others. Staggering the relative entry cycle times for the execution of a section of code provides a perfect method for hiding the delay imposed by high latency operations. For example, if the instruction that issues a high latency operation is scheduled on a processor that enters that section of code a sufficient number of cycles before the processor that uses the item, the effects of the latency will be hidden and parallelism can be exploited. In such a case it is possible that the *lead* PE will be executing instructions in a new section of code while *trailing* PEs are still completing previous sections.

The task of code separation is to separate the task across processing elements, with the goal of minimizing the effects of high memory latency and high functional unit latency for operations like multiply and divide by decoupling the *definition* of the data item from its *use*. Code separation is performed in two stages: code partitioning and load balancing.

4.5.1 Code Partitioning

The function of code partitioning is to organize the instructions specified in RTL form into a directed acyclic graph of *groups*. Much like initial register allocation strategies, code partitioning initially assumes an infinite number of processing elements. The actual partitioning algorithm begins by determining how individual instructions are initially grouped together. Strongly connected components are sequences of nodes in a directional graph which, for every pair of nodes v and w, there is a path from v to w and a path from w to v [5].

Each group consists of a sequence of mutually dependent RTL lines. The first task of the partitioning phase is to process the dependence graph in order to to identify instructions that are mutually dependent. Code partitioning examines the program dependence graph to find strongly coupled chains of instructions — those instruction sequences that have circular dependence chains. These chains can then be used to partition RTLs into dependence groups. The primary grouping, referred to as the *control group*, contains all branch operations and the lines in the RTL required to calculate branch conditions and targets (i.e. branch instructions and the instructions that they depend on). The branch instructions are duplicated for each dependence group in order to maintain a consistent control flow through the code (Alternate partitioning strategies can relax this condition if no data is being manipulated in a block by some PEs).

4.5.2 Load Balancing

Once the primary groups are partitioned into an acyclic graph, the load balancing phase determines how to partition the remaining groups. Load balancing is performed in two distinct stages: *Instruction Balancing* and *Group Fill*.

Instruction Balancing

Instruction Balancing tries to balance the amount of work performed by each PE. Each group is given a weight representing the expected number of cycles it will take to execute. This count is the product of the latency to execute the instructions in a group and the expected number of executions of the group. The set of all groups exceeding a certain weight is then partitioned across the available PEs. The latency measure can be calculated by assigning a latency value to each instruction type. The frequency measure can be determined by using previous profile information, or by static heuristic measures (e.g. groups in loops are executed more than groups outside of loops). With the reduction in the number of groups caused by the elimination of any groups that do not exceed the threshold weight, it is feasible to use an exhaustive search to determine the optimal partitioning.

[5] A standard graph algorithm for determining strongly connected components is used.

Once the optimal permutation is found for those groups that account for most of the execution time, the remaining groups are allocated as close to the lead PE as can be achieved without violating the uni-directional data flow restrictions between PEs. Later on, latency removal will be used to assign the remaining groups to the physical PEs.

Group Fill

The final stage in balancing instructions, *Group Fill*, builds the program dependence graph for the remaining (unallocated) groups and augments this graph with latency information. The groups are then sorted by their dependence relationship and each group is temporarily allocated to the latest (farthest from the lead) PE to which it can be assigned without violating the uni-directional data flow restriction placed on the schedule.

There are two aspects to group filling: Placing the remaining instructions across PE in a manner that does not lengthen the execution time of the most burdened PE and placing variables across PEs in a manner that does not overburden any one register file. In a machine with 128 general purpose registers (as a 4 processor MISC configuration has), register pressure is not a significant problem. In the initial register allocation, prior to code partitioning, 128 registers are assumed available to a single PE stream. During the code partitioning phase register limitations are ignored, leaving the final task of register assignment to the load balancing phase. The goal is to minimize the need to *spill* register contents to memory, and when spills are required, maintain the performance of the *lead* PE. This can be accomplished by scheduling the instructions required to implement the spill code on trailing PEs when possible.

The algorithm to perform the instruction load balancing examines the Directed Acyclic Graph (DAG) of dependence groups. A ready set of dependence groups is then identified and scheduled on the leading PE if the placement of this group does not overburden the execution resources of the lead PE (and thereby lengthen the instruction schedule). All groups left unallocated are considered for each successive PE until a schedule is found. The last PE accepts all remaining (unscheduled) groups. This is a greedy heuristic, but has the advantage of placing high latency operations across processing elements.

To illustrate the algorithms covered in the partitioning phase, the two example programs, InnerProduct() and LinkedList(), will again be utilized. The RTL form for these examples is shown in Figures 4.4 and 4.5.

In the InnerProduct() example program (Figure 4.2) the two independent memory operations (the vector loads for x and z) are split onto two processing elements. This leads to a schedule that utilizes the full capabilities of the target architecture. This schedule is shown in Figure 4.6.

```
[1]     t1 = 0            ; q=0
[2]     t2 = 0            ; k=0
[3]     t3 = 1024         ; set register for test
[4]     t4 = LOC[_z]      ; t4 = base of array z
[5]     t5 = LOC[_x]      ; t5 = base of array x

[6]     L1:
[7]     t6 = (t2>=t3)     ; calculate branch cond
[8]     PC = t6, L2       ; branch if true
[9]     t7 = M[ t4+t2 ] ; load t7= z[k]
[10]    t8 = M[ t5+t2 ] ; load t8= x[k]
[11]    t9 = t7 * t8      ; (z[k] * x[k])
[12]    t1 = t1 + t9      ; q = q + (z[k] * x[k])
[13]    t2 = t2 + 1       ; k++
[14]    PC = L1
[15]    L2:
```

Figure 4.4: Example Code: InnerProduct() RTL Form

4.6 Separation Strategies

This section examines the effect of various partitioning heuristics on the ability to separate the strongly connect groups across the available PEs in a particular MISC implementation. Partitioning and instruction balance are performed for a four PE MISC architecture.

4.6.1 Strategy 1: DAE Partitioning

The most restrictive form of separation partitions instructions in much the same manner as originally performed on the PIPE processor [51]. Each group is characterized as a CONTROL group, containing control and memory instructions, or FREE groups, containing all other instructions.

In the PIPE machine, the CONTROL group is executed on the *access* processor and the FREE groups are combined and assigned to the *execute* processor. This strategy is implemented in MISC by assigning all instructions in the CONTROL group to the *lead* PE and distributing the elements of the FREE groups among the remaining (*trailing*) PEs.

Figures 4.7 and 4.8 show the results of partitioning the intermediate code of the InnerProduct() and LinkedList() examples. Each line in the figures contains an enumeration (e.g. [1]), a group placement identifier and an RTL statement. The enumeration is used in the text to specify a line, the group placement identifier

```
[1]     t1 = head          ; list=head
[2]     t2 = 4             ; calc z offset from list
[3]     t3 = 8             ; calc x offset from list
[4]     t4 = 12            ; calc next offset from list
[5]     t5 = 0             ; set register for loop test
[6]     t6 = 0             ; value=0

[7]     L1:
[8]     t7 = (t1==t5)      ; calc loop branch cond
[9]     PC = t7, L2        ; branch if true (list==0)
[10]    t8 = M[ t1+t3 ]    ; load t8= list->x
[11]    t9 = (t8==t5)      ; calc inverse if condition
[12]    PC = t9, L3        ; branch if true (list->x==0)
[13]    t10 = M[ t1+t2 ];  load t7= list->z
[14]    t11 = t10 / t8     ; (list->z / list->x)
[15]    t6 = t6 + t11      ; value = value + (z[k] / x[k])
[16]    L3:
[17]    t1 = M[ t1+t4 ]    ; load list->next
[18]    PC = L1
[19]    L2:
```

Figure 4.5: Example Code: LinkedList() RTL Form

contains the partition group to which the RTL has been assigned and the RTL statement shows the intermediate form of the program. Each statement of the RTL description either defines a label or describes an operation to be performed in the resulting code; comments are delimited by ';'. Virtual register labels (specified as t1, t2, t3, etc.) define intermediate points in the calculation and may or may not map to physical registers or queues (for the sake of clarity, the actual register mapping has been omitted).

In the InnerProduct example (Figure 4.7, lines [8] and [14] are placed in the CONTROL group because they are branch instructions. Lines [9] and [10] are also placed in the CONTROL group because they are memory access instructions. Any RTLs that define a value later used by a CONTROL group operation (lines [8], [9], [10] or [14]) will also be placed in the CONTROL group to avoid moving data *upstream* (from a trailing PE to a leading PE); line [7] falls in this category because it defines the branch control value (register t6) used by the branch instruction in line [8]. Line [13] is also included because it calculates the loop iterator (k++). Finally, since no groups can be assigned to the FREE group that contain values used by the access group (no *upstream* communication allowed), lines [2] through [5],

```
     PE1 Code          PE2 Code          PE3 Code          PE4 Code
[1]                                                        t1 = 0
[2]  t2 = 0
[3]  t3 = 1024
[4]  t4 = LOC[_z]
[5]                    t5 = LOC[_x]

[6]  L1:               L1:               L1:               L1:
[7]  t6 = (t2>=t3)
[8]  PC = t6, L2       PC = t6, L2       PC = t6, L2       PC = t6, L2
[9]  t7 = M[t4+t2]
[10]                   t8 = M[t5+t2]
[11]                                     t9 = t7 * t8
[12]                                                       t1 = t1 + t9
[13]                   t2 = t2 + 1
[14] PC = L1           PC = L1           PC = L1           PC = L1
[15] L2:               L2:               L2:               L2:
```

Figure 4.6: Example Code: Balanced InnerProduct() RTL Form

which initialize the registers used by instructions in the CONTROL group, must also be assigned to the CONTROL group. This leaves three lines ([1], [11] and [12]) available to allocate to the FREE group. During execution of this example on a DAE partitioned processor, the CONTROL processor would execute 6150 instructions (accounting for 75 percent of all instructions), leaving 2049 instructions (25 percent) to be executed by the remaining PEs.

The LinkedList example shows similar results in the partitioning of its code. Most lines are allocated to the CONTROL group, while relatively few lines ([6], [14] and [15]) remain to execute on the FREE PEs. Assuming the linked list contains 1000 items, the dynamic count of instructions for the CONTROL processor is 8007 instructions (80 percent) and the FREE PEs process 2001 instructions.

At this point it is important to point out one of the limitations of this analysis. Clearly if 80 percent of all instruction executions are performed by a single PE (as in the LinkedList() example), it will be difficult to realize a high issue rate. However, it may be possible to achieve a substantial increase in performance if the original scalar schedule had very poor performance due to a few high latency operations and the partitioned code is able to schedule those high latency dependent operations between PEs. This would allow those latencies to be subsumed by the partitioning process (and the *slip* [52] between the asynchronous PEs).

Table 4.3 shows the results of DAE partitioning on the benchmark suite. The

```
[1]     FREE     t1 = 0           ; q=0
[2]     CONTROL  t2 = 0           ; k=0
[3]     CONTROL  t3 = 1024        ; set register for test
[4]     CONTROL  t4 = LOC[_z]     ; t4 = base of array z
[5]     CONTROL  t5 = LOC[_x]     ; t5 = base of array x

[6]              L1:
[7]     CONTROL  t6 = (t2>=t3)    ; calculate branch cond
[8]     CONTROL  PC = t6, L2      ; branch if true
[9]     CONTROL  t7 = M[ t4+t2 ] ; load t7= z[k]
[10]    CONTROL  t8 = M[ t5+t2 ] ; load t8= x[k]
[11]    FREE     t9 = t7 * t8     ; (z[k] * x[k])
[12]    FREE     t1 = t1 + t9     ; q = q + (z[k] * x[k])
[13]    CONTROL  t2 = t2 + 1      ; k++
[14]    CONTROL  PC = L1
[15]             L2:
```

Figure 4.7: Example Code: DEA Partitioning of InnerProduct()

first column identifies the application evaluated. Columns titled *Control* and *Free* show the partitioning of dynamic instruction executions into control and free groups (shown in thousands). The column titled *virtual* shows the maximum number of virtual PEs that are used in the partitioning algorithm: The Virtual PE entry gives a measure of how many PEs are required to partition the FREE groups (plus one for the CONTROL group) in a manner which minimizes the effects of high latency operations. The column titled *physical* shows the minimum number of PEs required to map the virtual PEs to a MISC implementation without degrading performance [6]. The Physical PE count shows the number of PEs required to achieve the highest performance; this will likely be less than the number of virtual PEs because scheduling can place some instructions in empty pipeline slots caused by high latency dependencies.

The most notable fact drawn from DAE partitioning is the heavy skewing of instructions toward the lead PE. While some performance improvement is found by decoupling the code in all but two benchmarks (eqntott and espresso), the benefits are marginal. In these applications very few instructions can be allocated to a secondary processor due to dependencies between control and access instructions.

Each application examined suffers from a poor partitioning of instructions using this strategy. None of the applications require more than two PEs because there are

[6]Note that this is not an optimal schedule, but does provide the same level of performance as schedules involving more PEs using the algorithms described in this dissertation.

```
CONTROL  t1 = head        ; list=head
CONTROL  t2 = 4           ; calc z offset from list
CONTROL  t3 = 8           ; calc x offset from list
CONTROL  t4 = 12          ; calc next offset from list
CONTROL  t5 = 0           ; set register for loop test
FREE     t6 = 0           ; value=0

         L1:
CONTROL  t7 = (t1==t5)    ; calc loop branch cond
CONTROL  PC = t7, L2      ; branch if true (list==0)
CONTROL  t8 = M[ t1+t3 ] ; load t8= list->x
CONTROL  t9 = (t8==t5)    ; calc inverse if condition
CONTROL  PC = t9, L3      ; branch if true (list->x==0)
CONTROL  t10 = M[ t1+t2 ]; load t7= list->z
FREE     t11 = t10 / t8   ; (list->z / list->x)
FREE     t6 = t6 + t11    ; value = value + (z[k] / x[k])
         L3:
CONTROL  t1 = M[ t1+t4 ] ; load list->next
CONTROL  PC = L1
         L2:
```

Figure 4.8: Example Code: DEA Partitioning of LinkedList()

plenty of excess cycles to hide latency in the second PE due to the heavy skewing of instructions. At this point, no load balancing strategy can achieve high levels of instruction level parallelism because the performance will always be dominated by the lead PE executing the great majority of the instructions. By scheduling 69% to 92% of all instructions on the lead PE, it will be impossible to obtain an execution rate greater than about 1.3 IPC (with a single issue lead PE).

However, this does not directly show the performance improvement that can be expected for an application. For instance, if a scalar implementation of an application has an execution rate of 0.3 IPC, then a MISC IPC of 1.2 is a great improvement even if it also has a relatively low IPC count. However, it is unlikely that the MISC design can be competitive with more advanced multiple issue architectures, if 80% of all instructions are executed in a strictly sequential manner. To help alleviate this problem a second partitioning strategy was developed that reduces the demands on the lead PE.

Table 4.3: DAE Partition Results (Dynamic)

Program	Control	Free	Virtual	Physical
compress	72	28	3	2
eqntott	83	17	6	1
espresso	92	8	3	1
gcc	76	24	6	2
xlisp	88	12	4	2
sc	69	31	8	2
povray	77	23	3	2
sobel	69	31	5	2
ear	76	24	8	2
KMP	60	40	4	3
InnerProduct	75	25	3	2
LinkedList	80	20	3	2

4.6.2 Strategy 2: Control Partitioning

The second approach to load balancing, *control partitioning*, does not require memory operations be placed on the lead processor unless they are required as part of a control group (i.e. they generate condition values used by conditional branches), which reduces the burden placed on the lead PE. Table 4.4 shows the results of this partitioning strategy on the benchmark suite.

Control partitioning shows improved results across all benchmarks. Four programs (sc, sobel, KMP and InnerProduct) achieve the best results when three or more PEs are available. Unfortunately, most of the integer benchmarks still show that little parallelism is exploited, with eqntott and espresso showing no improvement. These applications often use the result of a memory reference to determine the control flow; an example of this is traversing a linked-list, where the terminating condition (NULL) value can only be determined by retrieving the contents of memory for each item in the list. The scientific applications show a much better partitioning using this technique because their simple array access patterns are more easily partitioned; the memory access operations are now allocated to the FREE PEs, and in many cases there is little address aliasing to restrict the compiler from assigning references to more than one processor.

Examining our two example programs, we see improvement in InnerProduct(), but no improvement in LinkedList(). In the InnerProduct() example 4.9, the memory access operations (lines [9] and [10]) were successfully separated from the control group; they have no impact on the control flow decisions, so the control processor can continue processing independent of the execution of the load instructions. Lines

Table 4.4: Control Partition Results

Program	Control	Free	Virtual	Physical
compress	63	39	4	2
eqntott	82	18	6	1
espresso	92	8	2	1
gcc	56	44	9	2
xlisp	81	19	4	2
sc	52	48	8	3
povray	74	26	4	2
sobel	42	58	6	3
ear	65	35	8	2
KMP	47	53	6	3
InnerProduct	50	50	5	4
LinkedList	70	30	4	2

[4] and [5] can also be relocated because they do not generate values required by the CONTROL group. In fact, CONTROL partitioning is able to perform an optimal partitioning in this particular example. For this reason, it will be omitted throughout the remainder of the partitioning approaches.

As mentioned earlier, the LinkedList() example (Figure 4.10) shows little improvement in partitioning; only lines [2] and [13] could be relocated to the FREE group. The memory reference at line [10] returns data later used in a branch calculation, which forces the placement of that operation in the control group [7].

Finally, the memory access in line [13] can only be placed into the CONTROL group if the compiler can determine that it does not reference the same address as found in reference [10]. The vpcc compiler will make this decision (an unsafe one) because of the different element names in the structure. If this assumption cannot be made, no improvement over DAE partitioning will be found.

Applying control partitioning provides a more even distribution in the scheduled code, but the number of instructions placed in the CONTROL group still dominates. In addition, the instructions identified in the CONTROL column must be allocated to a single (the lead) PE. The next logical step in improving the partitioning is to relax the restriction on placement of control flow operations. By allowing individual PEs to follow independent routes through the control flow, branch points where very little computation is occurring can be assigned to a small number of PEs, freeing the rest to perform later computations. IF statements often have this

[7] It should be noted that the value read from memory by line [10] may be required by two PEs: The control PE to calculate the branch condition in line [11], and the division performed in line [14]. This is an example of a broadcast memory load operation.

```
[1]     FREE        t1 = 0
[2]     CONTROL     t2 = 0
[3]     CONTROL     t3 = 1024
[4]     FREE        t4 = LOC[_z]
[5]     FREE        t5 = LOC[_x]

[6]                 L1:
[7]     CONTROL     t6 = (t2>=t3)
[8]     CONTROL     PC = t6, L2
[9]     FREE        t7 = M[ t4+t2 ]
[10]    FREE        t8 = M[ t5+t2 ]
[11]    FREE        t9 = t7 * t8
[12]    FREE        t1 = t1 + t9
[13]    CONTROL     t2 = t2 + 1
[14]    CONTROL     PC = L1
[15]                L2:
```

Figure 4.9: Example Code: Control Partitioning of InnerProduct()

characteristic; it may be possible to have a trailing PE perform the entire computation of an if statement while the leading PEs can continue on after the completion of the conditional. Any point in the control flow graph that *joins* after a *fork* is a candidate for this modified partitioning scheme. This leads to the third partitioning strategy studied.

4.6.3 Strategy 3: Group Partitioning

The third approach requires only that those control groups containing CALL instructions be allocated to the lead processor. This is the least restricted partitioning algorithm examined that is completely supported by the current specification of the MISC architecture. This approach operates much like a very tightly coupled multiprocessor; not only are all PEs executing from a separate instruction stream, but they do not necessarily follow the same flow of control through the basic blocks. Some PEs may skip a block or many blocks. A restriction is still placed on the partitioning by requiring each PE to follow the same flow at function boundaries; if one PE enters a function, all PEs will enter the function. The results of performing group partitioning on the benchmark applications are shown in Table 4.5.

These results show that about half of the benchmarks are able to place enough instructions in the FREE groups to allow an even instruction distribution across a dual PE configuration. Two scientific applications, KMP and sobel, show excellent results using group partitioning. Group partitioning is also effective in the

```
CONTROL    t1 = head
CONTROL    t2 = 4
CONTROL    t3 = 8
CONTROL    t4 = 12
CONTROL    t5 = 0
FREE       t6 = 0

           L1:
CONTROL    t7 = (t1==t5)
CONTROL    PC = t7, L2
CONTROL    t8 = M[ t1+t3 ]
CONTROL    t9 = (t8==t5)
CONTROL    PC = t9, L3
FREE       t10 = M[ t1+t2 ]
FREE       t11 = t10 / t12
FREE       t6 = t6 + t11
           L3:
CONTROL    t1 = M[ t1+t4 ]
CONTROL    PC = L1
           L2:
```

Figure 4.10: Example Code: Control Partitioning of LinkedList()

LinkedList() example, Figure 4.11. The conditional branch controlling execution of the expression could be off-loaded from the control processor to another PE. This generates an improved partitioning which provides greater flexibility in balancing the load across four PEs. At this point, the linked list example has also been partitioned as well as possible (since the control group contains only a single strongly connected component). Further partitioning by analyzing memory conflicts will not alter the partitioning. However, if vpcc did not make the assumption that differing elements of a structure could not conflict, there would be no improvement possible at this point, until those conflicts could be resolved.

The majority of the integer applications still allow little opportunity to achieve balanced instruction issue. In these applications, much of the parallelism is in very tightly coupled groups of code that cannot be split across multiple PEs without violating the restriction on unidirectional data flow between PEs. Since this is the feature that allows decoupled processors to tolerate high memory latencies, relaxing that restriction could significantly reduce the advantages of decoupling.

One additional problem remains with finding enough groups to provide flexible partitioning. The semantics of the C language make it difficult to determine possible

Table 4.5: Group Partition Results

Program	Control	Free	Virtual	Physical
compress	60	40	4	2
eqntott	75	25	6	2
espresso	69	31	5	2
gcc	55	45	9	2
xlisp	52	48	5	2
sc	50	50	8	3
povray	42	58	7	3
sobel	30	70	7	4
ear	52	48	9	2
KMP	33	67	8	4
InnerProduct	50	50	5	4
LinkedList	40	60	5	3

memory conflicts at compile time. The freedom to assign a pointer variable to an arbitrary memory location makes it difficult to identify instructions that are likely to be independent, but cannot be absolutely determined to be independent. This requires additional links in the dependence graph which restrict the ability to allocate those memory operations to different dependence groups. The effect of this possible, but unlikely, dependence is to require a majority of all memory access operations to be placed on a single (lead) PE. A final partitioning strategy was studied which removes this problem from the partitioning algorithm.

4.6.4 Strategy 4: Memory Partitioning

All prior partitioning algorithms have had to schedule groups containing memory operations on a single PE unless it could be proved that no memory conflict was possible since function calls cross a barrier beyond which no analysis of memory references can be performed. A conservative approach to partitioning that restricts a balanced partitioning across PEs is therefore required.

The final approach studied allows memory groups to be partitioned in any fashion. This increases the ability of the partitioning algorithm to generate more FREE groups (and thereby improve later code balance). This strategy does require some support from the hardware and/or software to ensure correct memory ordering of operations in accordance with the specifications of the source program is maintained. This is a difficult task for the hardware to perform; however, it is vital to release the compiler from the tremendous restriction of scheduling in the presence of frequent function calls.

```
[1]     CONTROL    t1 = head
[2]     FREE       t2 = 4
[3]     FREE       t3 = 8
[4]     CONTROL    t4 = 12
[5]     CONTROL    t5 = 0
[6]     FREE       t6 = 0

[7]     L1:
[8]     CONTROL    t7 = (t1==t5)
[9]     CONTROL    PC = t7, L2
[10]    FREE       t8 = M[ t1+t3 ]
[11]    FREE       t9 = (t8==t5)
[12]    FREE       PC = t9, L3
[13]    FREE       t10 = M[ t1+t2 ]
[14]    FREE       t11 = t10 / t8
[15]    FREE       t6 = t6 + t11
[16]    L3:
[17]    CONTROL    t1 = M[ t1+t4 ]
[18]    CONTROL    PC = L1
[19]    L2:
```

Figure 4.11: Example Code: Group Partitioning of LinkedList()

An alternate approach would be to perform inter-procedural analysis to allow the compiler to determine the actual dependencies between memory instructions. The MISC compiler algorithms described in this chapter are being ported to the SUIF [53] compiler which is capable of such inter-procedural analysis [8]; this will provide a better platform for analyzing the actual capabilities of this approach.

Table 4.6 shows the effects of the Memory Partitioning approach assuming a perfect knowledge of memory conflicts can be determined during compilation. This technique shows a much greater ability to partition code in a manner that allows for a good load balance and good overall performance improvement. These results imply that, in languages without the pointer aliasing problem, or in a compiler with much better pointer analysis than vpcc, it is possible to achieve good code partitioning results. Many of the benchmark applications show a marked improvement in the ability to assign instructions to the FREE group, and show optimal PE assignments greater than two. This demonstrates that decoupled processors with more than two PEs are capable of providing improved performance.

[8] The version of SUIF incorporating inter-procedural analysis is not currently released outside of Stanford University. It should be included in a later release.

Table 4.6: Memory Partition Results

Program	Control	Free	Virtual	Physical
compress	34	66	6	3
eqntott	52	48	10	2
espresso	44	56	5	2
gcc	32	68	9	2
xlisp	39	61	7	3
sc	41	59	8	3
povray	42	58	8	4
sobel	30	70	7	4
ear	34	66	9	3
KMP	33	67	8	4
InnerProduct	50	50	5	4
LinkedList	40	60	5	3

4.7 Reducing Branch Duplication

Once the code has been separated across processing elements, branch instructions must be duplicated to enable each PE to follow the control flow of (its portion of) the execution. However, executing this many additional branch operations can potentially reduce the overall performance of the MISC architecture. Just as IF-conversion was used in section 4.2.2 to eliminate short forward branches, the MISC vector and sentinel instructions will be used to reduce the number of loop branches required during execution. At the same time, the hardware registers inside of MISC (e.g. VREG) can be used to calculate the loop induction variables. The instructions that calculate the induction variable may also be eliminated by translating using the hardware register (VREG) directly.

4.7.1 Using Vector and Sentinel Operations

Two optimization techniques are applied to loops in this compiler — *branch reduction* and *induction variable calculation*. These two optimizations attempt to eliminate instructions inside inner loops, which can lead to significant performance improvements when these loop iterations account for a large portion of the execution time.

For loops with few or no data dependencies between iterations, *loop unrolling* [54] is a popular technique to increase the efficiency of the list scheduler. During this optimization, iterations of the loops are explicitly expanded, making an increased number of instructions available to the scheduler. By providing additional (hopefully independent) instructions, the list scheduler is less likely to generate sparsely populated instruction streams.

For loops that cannot be efficiently unrolled, the MISC architecture provides two mechanisms to reduce the need for branch duplication: VLOOP/SLOOP instructions (sections 3.6 and 3.7) and predicated execution (section 3.8).

4.7.2 Induction Variable Calculation

An induction variable is a variable whose value is consistently modified (incremented or decremented) by a constant value on each iteration of a loop. These variables are often used to determine the number of iterations to be performed. Furthermore, induction variables are often used to index array data items or manipulate memory pointers, and can be defined in terms of an *induction expression*. While a number of expressions are possible, a common induction expression is:

$$IV_i = \begin{cases} dee & \text{if } i = 1 \\ IV_{i-1} + cee & \text{if } i > 1 \end{cases}$$

where i is the iteration count (value 1 on the first iteration), *cee* is the amount by which the induction variable is incremented during each iteration of the loop, and *dee* is the value of the induction variable at the start of the first iteration. The detection of induction variables is a well understood problem. The algorithm used in this compiler is derived from [30] (Algorithm 10.9).

Once the control state of the machine has been extended to support loop operations, it is a simple modification to handle the calculation of induction variables used in the loop. The *src1* and *src2* fields of the VLOOP instruction are free to contain the *cee* and *dee* values; *VREG* will maintain the induction value and *src3* will control loop termination as described above.

In the example of InnerProduct() in Figure 4.6 both array index calculations can be performed by the hardware using this technique. Using this technique leads to the RTL description after loop translation shown in Figure 4.12.

4.8 Instruction Scheduling

The final stage of the compilation process schedules the instructions on each individual PE. A least cost schedule is developed that attempts to schedule all instructions to execute in the shortest time. List scheduling on MISC operates on each of the processing element individually, scheduling to avoid wasted cycles (due to latency). Simple list scheduling is complicated by the need to interpret queue register specifications in the RTL and to avoid reordering queuing operations. Furthermore, all loop instructions (VLOOP and SLOOP) are examined to determine the number of instructions in the delay slots; if only one instruction is iterated in the loop,

```
     PE1 Code          PE2 Code          PE3 Code          PE4 Code
[1]                                                         t1 = 0
[2]
[3]  t3 = 1024         t3 = 1024         t3 = 1024          t3 = 1024
[4]  t4 = LOC[_z]
[5]                    t5 = LOC[_x]

[6]
[7]  t6 = (t2>=t3)
[8]  VLOOP 1,0,t3      VLOOP 1,0,t3      VLOOP 1,0,t3       VLOOP 1,0,t3
[9]  t7 = M[t4+VREG]
[10]                   t8 = M[t5+VREG]
[11]                                     t9 = t7 * t8
[12]                                                        t1 = t1 + t9
[13]
[14]
[15] L2:               L2:               L2:                L2:
```

Figure 4.12: Example Code: Loop Transformation of InnerProduct()

then the loop instruction is removed and the scalar instruction residing in the delay slot is translated into vector (or sentinel) form. In the InnerProduct() example in Figure 4.12, the VLOOP operations for each of the processors can be replaced with a single vector instruction since each loop consist of only one instruction and the default induction calculation is used (or no induction variable is referenced in PE3 and PE4).

4.9 Summary

This chapter has described the structure and important characteristics of the MISC compilation process. The effects of various partitioning strategies were examined to assess the feasibility of translating high-level application code to a decoupled implementation.

The partitioning results indicate that while some parallelism can be extracted using a DAE model under these compilation techniques, dramatic improvements in ILP are unlikely because of the large number of instructions executed by the lead PE. More aggressive partitioning techniques can improve performance, especially in those applications using simple, non-pointer based data structures. Instruction level parallelism exploited by a decoupled approach exploits different types of parallelism as that of a superscalar design. This suggests that a hybrid solution in which each

MISC processing element implements a superscalar design could provide greater issue widths and improved performance. Extending this analysis to evaluate total execution time and the ability to hide memory latency between processing elements will be done in the next chapter.

Chapter 5

Performance of the MISC Architecture

This chapter examines how well the MISC system performs executing a series of benchmark applications. The experiments are separated into three parts. First, the performance of the MISC system is measured using kernels of scientific code and results are compared to previous decoupled [55] and superscalar [56] designs. These kernels contain highly parallelizable loops whose ILP can be efficiently exploited by a variety of ILP models. The second set of experiments measures the ability of the MISC architecture and compiler to hide the high latency memory operations using the partitioning techniques introduced in chapter 4. Finally, execution time for the MISC system is measured for both integer and scientific applications and compared to a current superscalar design.

5.1 Performance Results on Livermore Loop Kernels

The Lawrence Livermore Loops were selected as the benchmarks for comparing code separation on a four PE MISC architecture with optimized compilation of MIPS code. The loops were extracted from large applications used at the Livermore National Laboratory. These loop *kernels* are best used to study the performance capabilities of different supercomputer designs in extracting parallelism for scientific applications. The first 12 loops were compiled for both the MIPS and $MISC_4$ architectures (the MIPS code was compiled using the cc compiler with optimizations enabled).

In order to compare the performance of the $MISC_4$ and MIPS processors the total number of cycles required to complete a given loop was measured. This is an obvious method of evaluation when dealing with conventional architectures, but misses some of the capabilities that exist in the multiple instruction stream approach. Higher latencies were used for the $MISC_4$ architecture because the design anticipates a higher clock speed.

The performance of the MIPS architecture in executing these applications was calculated by hand, while the $MISC_4$ architecture was studied using a behavioral level simulator executing the binary (assembled) form of each loop. Both the MIPS and $MISC_4$ simulations assumed a perfect ICache and an 8K-byte, 32-byte line UCache configuration. The operational latencies used in the simulations are shown in Table 5.1.

Table 5.1: Operational Latency

Operation Unit	MIPS	$MISC_4$
memory load	2	10
Integer Add	1	2
Int Mult	2	4
Branch	2	2

Table 5.2 shows the results of executing the Livermore loops on MIPS and $MISC_4$ architectures. The MIPS configuration was capable of issuing only a single instruction each clock cycle, while the $MISC_4$ system consisted of four PEs each capable of issuing a single instruction per cycle. The first column identifies each of the 12 loops studied. The second column shows the number of clock cycles required to execute that loop on the $MISC_4$ system. The third column shows the number of execution cycles on a MIPS (scalar) processor. The fourth column shows the performance improvement (calculated as MIPS cycles divided by $MISC_4$ cycles) achieved by the $MISC_4$ design.

For a majority of the loops we see a three to four-fold decrease in $MISC_4$ cycle counts. This demonstrates that much of the parallelism available in these benchmarks is effectively being extracted by the $MISC_4$ design. However, several of these benchmarks show less of a performance increase than others (most notably LLL6 and LLL11). This is due to a recurrence constraint found in the data manipulated by the loop — in these cases, adding more processors will not increase performance regardless of the approach used, since the parallelism is simply not available in the loop.

To provide a comparison with a similarly configured single instruction stream, multiple issue architecture these loops were hand compiled for a 4 issue VLIW architecture. The VLIW architecture chosen is based upon the most sophisticated version found in [57]. This VLIW machine allows four instructions to be issued per clock cycle and there are no limitations on the type of instructions that can be issued. The register file is capable of handling eight read and four write requests on each cycle, and perfect register renaming (using a rotating register file [57]),

Table 5.2: LLL Comparison: MIPS vs $MISC_4$

Benchmark Loop	MIPS (cycles)	$MISC_4$ (cycles)	Performance Improvement
LLL1	5611	1232	4.55
LLL2	1112	256	4.34
LLL3	6664	2063	3.23
LLL4	3011	753	3.99
LLL5	6979	1994	3.50
LLL6	7726	4982	1.55
LLL7	4338	859	5.05
LLL8	3218	1476	2.18
LLL9	4081	813	5.02
LLL10	3107	1007	3.08
LLL11	3049	2003	1.52
LLL12	3759	1013	3.71

predicated execution, and loop control operation are assumed. Furthermore, it is assumed that induction variable calculations can be included in any memory reference operation (providing pre/post increment capability). The loops were hand compiled due to a lack of a VLIW compiler capable of supporting the hardware specifications above. Modulo Scheduling was used to provide software pipelining to minimize latency delays, and to exploit the other hardware capabilities of the architecture.

It should be noted that for these highly regular loops, and assuming fixed latency operations, this idealized VLIW architecture will perform as well as *any* other 4-issue architecture (superscalar and decoupled included), excluding initialization effects. Comparing the four PE $MISC_4$ system to this VLIW design shows how close the MISC approach comes to an optimal design for these loop kernels.

The results in Table 5.3 show that the MISC approach is capable of extracting virtually the same level of parallelism in most of the loops as this idealized VLIW machine. In addition, the MISC design has an additional benefit unavailable to the VLIW machine; since some of the lead PE(s) terminate a loop before trailing PEs, they are capable of starting the execution of the code following the loop exit early. This demonstrates an important point in the evaluation of the performance capabilities of a multiple instruction stream processor; where all functional units in the VLIW processor are locked into the loop (even if they have nothing to do), the $MISC_4$ processor requires only those PEs necessary to execute the loop code, while the remaining PEs can immediately start the execution of the next code block(s).

The last two columns in the table show the code improvement of the $MISC_4$

machine over the VLIW. The first of these values indicates how soon each machine was able to start execution of the instructions following the loop. The second value shows the relative completion times of the loop.

Table 5.3: LLL Comparison: $MISC_4$ vs VLIW

Bench	PE1	PE2	PE3	PE4	$MISC_4$	VLIW	Improve in first PE	Improve in last PE
LLL1	1205	1215	1221	1232	1232	1236	1.35	1.00
LLL2	201	201	211	256	256	228	1.13	0.89
LLL3	1025	1025	1035	2063	2063	1065	1.01	1.00
LLL4	385	395	404	753	753	771	2.00	1.02
LLL5	997	999	1993	1994	1994	1994	2.00	1.00
LLL6	0	997	1995	4982	4982	4984	∞	1.00
LLL7	727	846	736	859	859	863	1.18	1.01
LLL8	586	720	1240	1476	1476	1456	2.48	0.99
LLL9	609	707	712	813	813	708	1.16	0.87
LLL10	506	506	1006	1007	1007	1014	2.00	1.01
LLL11	0	999	1000	2003	2003	2004	∞	1.00
LLL12	1002	1012	1013	1013	1013	1013	0.99	1.00

To demonstrate the effects of this let us examine two of the loops in more detail. An examination of the execution of LLL6 reveals that only the final processing element is required to perform the majority of the loop calculation. This is due to a tight recurrence relation found in the loop equation. In the VLIW machine all functional units are forced to sit idle in the loop body until the aggregate machine completes calculation of the loop. In the $MISC_4$ approach, the three processing elements not involved in the recurrence calculation are free to continue executing subsequent code.

If we now assume that LLL11 follows the execution of LLL6, we can determine the different stagger rates on exit from LLL6 and reschedule LLL11 to take advantage of the free processing elements. Table 5.4 shows the result of this rescheduling (done at compile time) and compares it to the idealized VLIW architecture. As seen in the table, the ability to overlap execution of the loops allows the $MISC_4$ processor to perform both loops in the time required by the VLIW architecture to perform the first alone.

This result demonstrates the advantage that the multiple instruction stream attains across basic blocks. Since each PE enters and leaves any block of code in a manner that is asynchronous with respect to the other PEs, it enables the lead PE to continue with later processing before the trailing PEs complete the earlier computation. This means that the high latency communication originating from

Table 5.4: LLL Comparison: $MISC_4$ vs VLIW of LLL6-11

Bench	PE1	PE2	PE3	PE4	MIPS	VLIW	Improve	Improve
LLL6	0	997	1995	4982	4982	4984	∞	1.00
LLL11	999	1000	2003	0	2003	2004	∞	1.00
LLL6-11	999	1997	3998	4982	4982	6988	6.99	1.40

the lead PE and destined for trailing PEs is not only amortized within a block of code, but also throughout a function or program; once the lead PE gets ahead, it stays ahead even through artificial boundaries (like a basic block).

5.2 Evaluating $MISC_4$ Performance on Complex Applications

In this section some performance characteristics of the $MISC_4$ processor will be compared to both a scalar and a superscalar design. Various components will be evaluated, including the affects of various load balancing mechanisms, the affect of load latency and the memory organization.

The simulation environment used in this study differs slightly from that used in the previous section in that complete runs of the SPECint and scientific applications (described in chapter 4, Table 4.1) are simulated, including all library routines.

5.2.1 Simulation Environment

In order to execute large applications on a new architecture, such as MISC, the standard C library (as well as several others) must be compiled and system support included in the simulator. This is a significant task for a new architecture which has only a compiler and assembler and no operating system. An alternate approach is to perform a detailed simulation of the architectural features within the framework of an existing architecture. This second alternative was chosen for this study and the Alpha processor [58] was selected as the simulation platform. The behavioral simulation of MISC instructions has been incorporated into the ATOM [59] analysis tool set; Alpha binaries are now executed, but behave as if they were running on a $MISC_4$ system. Using this approach also requires that portions of the compiler (e.g. code partitioning) be incorporated into the ATOM simulator. This was done as follows:

1. Compile the benchmark applications for an Alpha system, generating an Alpha object file.

2. Link the object file(s) with the standard library and any other required libraries generating an executable version of the program.

3. Translate the executable into an Intermediate Representation (IR) that can be processed by those components of the MISC compiler responsible for code partitioning and load balancing.

4. Annotate the IR with information about the $MISC_4$ schedule and the original executable instructions.

5. Simulate the execution of the program under ATOM simulating the data and control flow for the $MISC_4$ processor using the annotations. This entails incorporating most of the MISC simulator into an ATOM application to determine the event timing during the execution.

6. Gather the simulated performance measurements for analysis.

5.2.2 Caveats

There are a number of limitations to this method of analysis. Register allocation cannot be performed, so the large number of registers available in the MISC architecture cannot be exploited. This will limit the aggressiveness of the optimizing transformations (at least those that trade increased register use for a scheduled performance gain).

A second limitation is the inability of the simulator to provide accurate instruction cache behavior. Therefore, for the purpose of this study the instruction cache is assumed to contain all necessary instruction data (i.e. the ICache access always *hits*). This is unlikely to affect the results significantly because a relative comparison is being made, and each processor simulated will assume the same instruction cache configuration.

Finally, the data cache will process references in the order that would be seen in the original Alpha execution. While this will yield a valid $MISC_4$ reference permutation, it will not simulate the asynchronous behavior of the MISC memory design. However, the $MISC_4$ UCache is modeled so references can be re-ordered during simulation, resulting in reasonable cache performance even if the original ordering in the Alpha schedule is poor. ICache access time is 1 cycle for all MISC and Alpha configurations. UCache and Alpha Data Cache access times are assumed to be 3 cycles while memory latency is 100 cycles. The UCache and Alpha Data Cache are configured as 64Kbyte 4way interleaved caches with 128 byte direct mapped lines. The operational latencies used in the simulations are shown in Table 5.5.

Table 5.5: Operational Latencies

ICache hit	1 cycle
ICache miss	not simulated
UCache hit	3 cycles
Memory load	100 cycles
Integer Add	1
Int Mult	4
Branch	1

5.3 Hiding Memory Latency

The latency of memory is increasing relative to CPU speed and will continue to do so for the foreseeable future. Larger cache structures can help alleviate this problem to some extent, but processor designs capable of tolerating high memory latencies will eventually be required. If we examine the latency tolerance of a superscalar design without a cache, we see that any performance gains due to increased instruction processing capability are overshadowed by memory stalls.

Table 5.6 shows the ability of each of the four previously described partitioning strategies to hide memory latency using inter-PE transfers in the integer SPEC benchmark applications. The percent of total memory latency is shown in each cell; total memory latency is calculated by accumulating the latency for all memory load operations (between the access request and the placement of the data into the MQ of one or more of the destination PEs). Latency figures are shown for both intra-PE and inter-PE memory requests. The third column shows the percent of memory latency comes from memory loads which originate and are destined for PE1 . Intra-PE latency will slow the application down while the data is retrieved; ideally, inter-PE latency should not affect overall performance. Results are shown for each benchmark application (identified in column one) using each of the partitioning strategies discussed previously in chapter 4 (identified in column two).

Table 5.6: Hiding latency using inter-PE queues for SPECint benchmarks

Program	Partition Strategy	Latency in PE1	Latency in PE2	Latency in PE3	Latency in PE4	Inter-PE Latency
compress	DAE	78	0	0	0	22
	Control	63	4	0	0	33
	Group	63	4	0	0	33
	Memory	41	8	2	0	49
eqntott	DAE	100	0	0	0	0
	Control	100	0	0	0	0
	Group	81	5	0	0	14
	Memory	67	3	0	0	30
espresso	DAE	100	0	0	0	0
	Control	100	0	0	0	0
	Group	43	12	0	0	45
	Memory	27	21	0	0	52
gcc	DAE	74	6	0	0	20
	Control	66	14	0	0	20
	Group	66	14	0	0	20
	Memory	35	22	0	0	43
xlisp	DAE	88	0	0	0	12
	Control	88	0	0	0	12
	Group	53	10	0	0	27
	Memory	42	6	8	0	44
sc	DAE	73	1	0	0	26
	Control	67	9	0	0	24
	Group	67	9	0	0	24
	Memory	50	2	10	0	38

The table shows that much of the memory latency cannot be hidden between PEs (shown in fifth column) but resides in the lead PE for most strategies. The DAE strategy has little opportunity to hide much of the latency since the vast majority of instructions must be located on PE1. The Control strategy and Group strategy results show some improvement by reducing the burden on PE1 — particularly the ability to move the branch calculation of some short forward branches (usually generated by IF-THEN-ELSE statements) from PE1 to other PEs. The ability to hide the latency of memory operations is still limited by the inability of the compiler to guarantee that different memory operations (LAQ/SAQ instructions) do not conflict. The Memory Partitioning Strategy shows that when memory access requests are allowed to be distributed between PEs, much of the latency can be hidden in the inter-PE queues.

Table 5.7 presents the memory latency results for the set of scientific applications. These applications show that a much greater percentage of memory operations can be subsumed in inter-PE queues. This is primarily due to the use of less abstract data types; simple array references dominate as opposed to the trees and linked-lists in the SPECint applications.

Table 5.7: Hiding latency via inter-PE queues for scientific benchmarks

Program	Partition Strategy	Latency in PE1	Latency in PE2	Latency in PE3	Latency in PE4	Inter-PE Latency
povray	DAE	75	0	0	0	25
	Control	75	0	0	0	25
	Group	35	19	0	0	46
	Memory	35	19	0	0	46
sobel	DAE	40	0	0	0	60
	Control	19	0	0	0	81
	Group	8	0	0	0	92
	Memory	8	0	0	0	92
ear	DAE	88	0	0	0	12
	Control	76	6	0	0	18
	Group	52	10	7	0	31
	Memory	40	9	7	0	44
KMP	DAE	62	0	0	0	38
	Control	32	0	0	0	68
	Group	8	9	3	0	80
	Memory	8	9	3	0	80

Separating memory access operations improves the partitioning of the code enough to allow a majority of the memory latency to be hidden in two of the applications. Another type of application, as seen in the espresso benchmark, benefits from removing the restriction on partitioning branch operations. Unlike the SPECint benchmarks, little additional improvement is found when employing the Memory partitioning strategy — these applications do not require sophisticated pointer analysis to determine whether memory conflicts may occur between access instructions.

The results for the scientific codes agree with those of the integer benchmarks, showing that the most restrictive partitioning strategies are only able to hide a small percentage of high latency memory operations between processing elements; the exception to this is the sobel benchmark, in which execution time is dominated by a simple loop with no intra-loop control flow.

5.4 Execution Performance

The final test of MISC performance is to examine total execution time compared to existing architectures. It should be noted that the SPECint benchmarks represents the worst case for decoupled design with their heavy instruction interdependence. Three different configurations of the Alpha architecture are compared to two and four PE versions of the MISC processor, $MISC_2$ and $MISC_4$ respectively. The Alpha was selected because it is a simple superscalar design that is available in two and four issue implementations. Each Alpha implementation performs in-order instruction issue and can issue one, two or four instructions during each cycle as long as all dependencies can be resolved at instruction issue. These configuration are labeled $Alpha_1$, $Alpha_2$ and $Alpha_4$ respectively. This models several hypothetical implementations that are less restrictive than current Alpha 21064 [58] or 21164 [60] implementations. Both the Alpha designs and the MISC design use a two level cache and a 50 cycle access time to main memory. Table 5.8 shows the total execution time of the MISC system relative to the scalar $Alpha_1$ system.

Table 5.8: Relative execution time for scalar, superscalar and MISC designs for SPECint benchmarks

Program	Partition Strategy	$Alpha_1$ Processor	$Alpha_2$ Processor	$Alpha_4$ Processor	$MISC_2$ System	$MISC_4$ System
compress	DAE				1.11	1.11
	Control				1.29	1.29
	Group				1.31	1.31
	Memory	1.00	1.54	1.95	1.52	1.93
eqntott	DAE				1.00	1.00
	Control				1.00	1.00
	Group				1.13	1.13
	Memory	1.00	1.71	2.01	1.64	1.78
espresso	DAE				1.00	1.00
	Control				1.00	1.00
	Group				1.50	1.72
	Memory	1.00	1.34	2.43	1.89	2.12
gcc	DAE				1.21	1.21
	Control				1.28	1.31
	Group				1.28	1.31
	Memory	1.00	1.56	1.98	1.64	1.92
xlisp	DAE				1.16	1.16
	Control				1.20	1.20
	Group				1.45	1.67
	Memory	1.00	1.39	1.92	2.05	2.48
sc	DAE				1.23	1.23
	Control				1.34	1.34
	Group				1.37	1.45
	Memory	1.00	1.49	1.84	1.46	1.75

The results show that with enough memory aliasing, the performance of a decoupled machine is similar to the performance of a simple superscalar design with the same issue width. In some cases, the ability to hide memory latency allowed further improvements.

Table 5.9 shows the total execution time for each of the scientific applications. The results for these applications differ significantly from those of the SPECint benchmarks. In three of the applications, both two-issue and four-issue versions of the MISC architecture are able to exploit more parallelism than the corresponding superscalar architectures. This is due to the ability of the ACCESS PEs in a decoupled design to continue initiating further requests to the cache when an in-order superscalar design would block soon after a cache miss. This is most clearly seen in the KMP application where the MISC system achieves twice the ILP of the corresponding superscalar approach.

Table 5.9: Relative execution time for scalar, superscalar and MISC designs for scientific benchmarks

Program	Partition Strategy	$Alpha_1$ Processor	$Alpha_2$ Processor	$Alpha_4$ Processor	$MISC_2$ System	$MISC_4$ System
povray	DAE				1.21	1.21
	Control				1.21	1.21
	Group				1.83	2.54
	Memory	1.00	1.72	2.21	1.83	2.54
sobel	DAE				2.11	2.11
	Control				2.41	2.73
	Group				2.76	3.04
	Memory	1.00	1.72	2.20	2.76	3.04
ear	DAE				1.13	1.13
	Control				1.27	1.27
	Group				1.32	1.65
	Memory	1.00	1.75	2.07	1.62	1.92
KMP	DAE				1.42	1.42
	Control				1.73	1.73
	Group				3.14	3.91
	Memory	1.00	1.52	1.83	3.14	3.91

5.5 Summary

Two distinct experiments were performed in this chapter: the evaluation of the limits of the MISC design in exploiting parallelism for highly parallelizable loop kernels, and an evaluation of the ability of both the architecture and the MISC compiler to identify and exploit the parallelism available in a broader class of applications.

The Livermore Loop kernels were used in the first experiment and showed that the MISC design is capable of extracting the same level of parallelism as an idealized VLIW implementation, with the added benefit of a more dynamic instruction execution around loop boundaries. Overall the MISC design not only achieved the same level of performance on individual kernels, but allowed for an overlap of execution when multiple kernels were scheduled to run in sequence.

The second set of experiments involved more general applications comprised of programs from the SPEC92 benchmark suite as well as a few additional applications from a variety of scientific domains. These results show that the ability of a decoupled design to identify and exploit the available parallelism depends on the code partitioning strategy of the compiler; a restrictive partitioning algorithm, such as the DAE strategy, does not allow for sufficient flexibility in the instruction schedule to exploit the parallelism available in the application. More flexible partitioning strategies do allow for efficient exploitation of parallelism achieving better performance to that of comparable superscalar designs. We intend to extend this analysis to compare a modified version of MISC with the more aggressively scheduled superscalar designs that are becoming more common.

The ability to hide memory latency between asynchronous processing elements was also studied. Again, when the compiler implemented a sufficiently flexible partitioning strategy, the architecture was capable of hiding much of the memory latency.

Chapter 6

Improving Memory Performance

In the previous chapters, the MISC architecture and compiler model were described and evaluated. The performance of the system has been shown to be quite good when compared to similarly configured superscalar designs. However, in order for any application to be able to achieve a high level of performance, the memory system must be capable of supporting multiple references each cycle. The MISC design provides unique opportunities for constructing a very high performance memory system; the use of decoupled memory access instructions and memory buffers permits more requests to be sent to the memory system per cycle. By providing more outstanding memory requests, the MISC cache can potentially improve the ordering of requests to maximize throughput.

This chapter measures some performance characteristics of a cache design supporting several of the features present in the MISC UCache. While the MISC UCache is used throughout this analysis, the results are equally applicable to any high performance processor design capable of supporting multiple outstanding memory requests.

6.1 Performance of the Cache

As described previously, the ability to handle multiple references during each cycle is necessary in order to expand the amount of instruction level parallelism that can be exploited by a processor. However, the design of a cache capable of supporting multiple references can pose a considerable challenge to the hardware engineers. Several alternative implementations are possible; one approach is to duplicate resources (as done in the register file). A multi-ported register file trades silicon area for the ability to handle multiple references per cycle, and is used because of the relatively small number of registers that exist in most CPU implementations. However, using the same tradeoff for a cache design is problematic. The number of elements assigned to the cache unit usually dwarfs that of the register file to begin with, so duplicating structures would result in a potentially unacceptable increase in transistor requirements (or an equally unacceptable decrease in cache size). This section explores some of the features of the MISC UCache approach that allows it to maintain the level of performance of a true multi-ported design using a simplified cache implementation which removes the need to duplicate cache memory.

While the design of the UCache is heavily influenced by the decoupled nature of the MISC architecture, some of the features found in this cache are equally applicable to many of the recent aggressive superscalar designs. The MIPS R10000 [63], in particular, implements a two-way interleaved cache similar to the MISC interleaved UCache and should benefit from the reordering [61] and combining [62] capabilities in the MISC design. The remainder of this section analyzes the performance of a MISC UCache-type design and compares it to a conventional multi-ported approach.

6.1.1 Experimental Configuration

To analyze the performance potential of various multiple access cache organizations, six cache models were examined, ranging from an interleaved model which is easy to implement to a fully multi-ported design capable of processing multiple references without restriction. The six configurations are:

1. *I*: a four-way interleaved cache [1] consisting of four 16K byte direct mapped caches. **(I)**
2. *IR*: a four-way interleaved cache with a reorder buffer on each address buffer. Each bank consists of a 16K byte direct mapped cache. **(I + reorder)**
3. *IC*: a four-way interleaved cache with combining support. Each bank consists of a 16K byte direct mapped cache. **(I + combining)**
4. *IRC*: a four-way interleaved cache with a reorder buffer and combining support. Each bank consists of a 16K byte direct mapped cache. **(I + reorder + combining)**
5. *MP*: a four-ported, 64K byte direct mapped cache. **(MP)**
6. *MPR*: a four-ported, 64K byte direct mapped cache with a reorder buffer. **(MP + reorder)**

These configurations have been chosen to see how well reordering and combining can be incorporated into an interleaved and/or multi-ported configuration, and how the performance of these configurations matches that of a conventional multi-ported cache.

[1] The bits of the effective address that are adjacent to those of the cache lines selector are used to select the cache bank.

An external memory access cost of 50 clock cycles (on a cache miss) is simulated. It is further assumed that sufficient bandwidth between the cache and external memory is available to guarantee that delays due to insufficient bandwidth will not occur. True dependencies in the application program may still limit the number of memory operations which may be outstanding at any cycle, since some of these applications simply do not have sufficient parallelism to fully exercise a high performance cache. An extreme example of this is found in the linked list example in chapter 4, in which it was impossible to perform memory fetch operations from different iterations due to the dependency between the value loaded in one iteration and the effective address calculation in the next iteration.

6.1.2 Reordering Memory Requests

Reordering of memory operations is possible when the order of the requests does not affect the values calculated during execution. This is easily determined in a sequential application by examining the effective addresses of the requests; if two references do not refer to the same location, then the order in which the cache processes these requests is not important to the correct execution of the program. Reordering memory requests allows memory load operations to bypass stores allowing the latency of the load operation (critical to the system performance) to be reduced. It is also often advantageous to allow loads to bypass each other, and since no memory state modification occurs with a load operation, no conflict test is necessary. Finally, it may be useful to allow stores to bypass each other; this occurs when the interleaved cache cannot process the first store because of a bank conflict with an earlier reference, but a later store maps to a free cache bank. By allowing the store operations to be reordered, multiple cache accesses can be supported.

Reordering of requests can also improve the performance of an interleaved cache by reducing the likelihood of bank conflicts. This is particularly important because locality in the reference pattern increases the probability that consecutive references will map to the same cache bank. Figure 6.1 shows the probability of a bank conflict between consecutive cache references in a four-way interleaved cache design. Each column in the figure shows the probabilities for a different SPEC application. The columns are separated into four segments, corresponding to the four banks in the cache. The lowest segment (*same bank* entry) shows the probability that for any reference the immediate successor will map to the same bank (causing a conflict). The other three segments show the probability that the immediate successor maps to each of the other banks in the cache.

If the reference steam contains independent (and random) references, the probability for each of the four segments should be evenly distributed (25 percent each). However, most applications show a skewed probability towards bank conflicts — averaging 38 percent across all applications and as much as 50 percent in the compress benchmark. This clearly limits the effectiveness of an interleaved approach

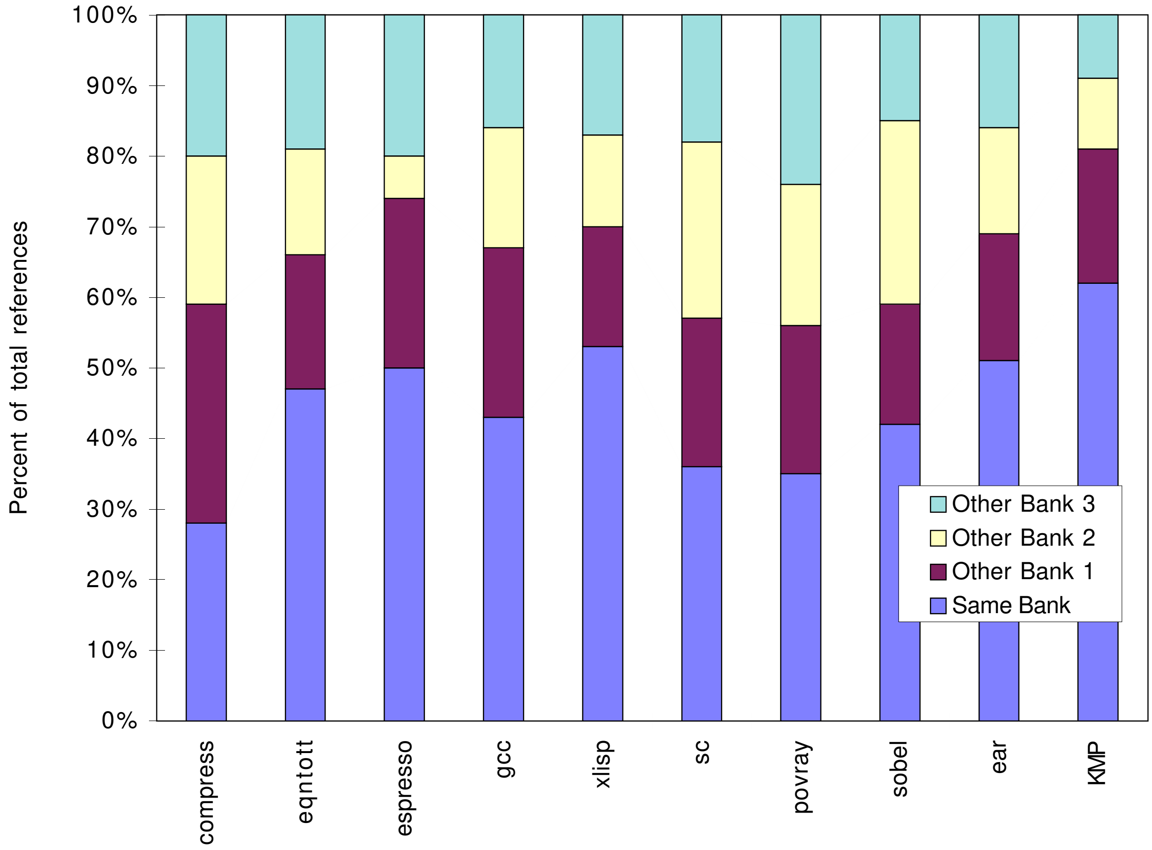

Figure 6.1: Relative distribution of cache bank references

when no reordering logic is provided.

The technique of reordering memory operations has been explored by Smith [9] and has more recently been implemented in the R10000 processor. These designs show a significant improvement in cache performance, especially when cache misses can be bypassed (a technique referred to as a NON-blocking cache design [64]). The performance impact of reordering references is a function of the number of outstanding requests, the restrictions on cache access (bank selection policy) and the nature of memory requests in the application. The effectiveness of reordering logic is improved when there are more outstanding memory requests. Decoupled architectures are designed to maximize the number of outstanding memory request by placing the requests on an ACCESS processor that runs ahead of the use of the data. This improves the effectiveness of the reorder buffer.

Figure 6.2 shows the effects of reordering on a bank selected cache for the integer benchmark applications described in Table 4.1. Four of the previously described configurations are included (I, IR, MP, MPR); the configurations employing combining logic will be examined in the next section. The average access rate for each configuration was measured across all benchmark applications [2]. The vertical axis shows the average number of references that can be processed by the cache in each cycle. The horizontal axis shows the affect of scaling the size of the reorder buffer on the access rate. Reorder buffer size ranges from 4 to 16 entries.

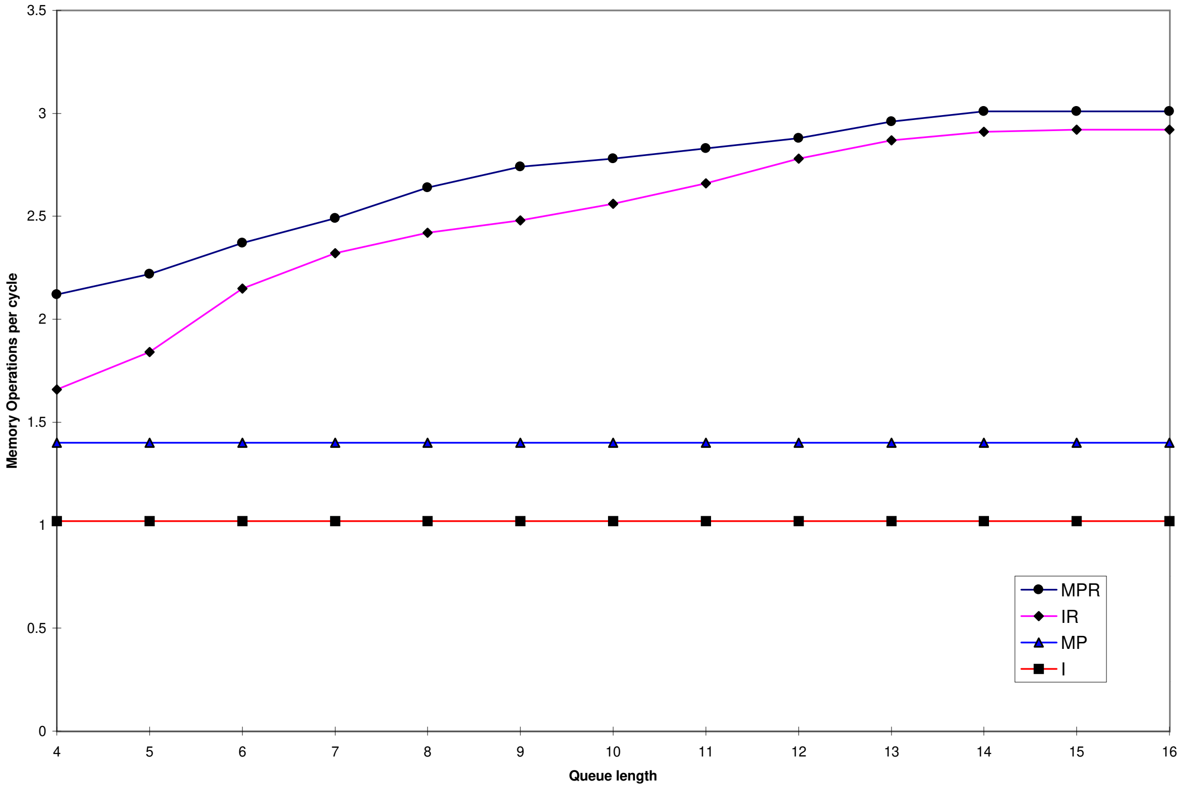

Figure 6.2: MISC Cache: Effects of Queue Length

It can be seen from the figure that no configuration initially approaches the maximum possible reference per cycle rating of the cache (4 references per cycle). Two effects limit the ability to locate additional independent references; miss processing and interdependence between load instructions. When several cache misses occur, the reorder queue fills up with those requests that are being processed through external memory. This limits the number of available outstanding memory requests

[2]Each benchmark application was weighed equally in determining the average so that benchmarks with a longer reference stream did not dominate the average.

to those that can fit in the remaining (free) queue entries. With a 50 cycle latency to retrieve data on a cache miss, it is likely that most (and occasionally all) of the reorder slots will be filled with unprocessed cache misses. The figure shows that a reorder buffer of length four provides a 60% improvement in cache throughput for both interleaved and multi-ported configurations.

As the queue size is increased, sufficient queue entries exist to store numerous cache misses while retaining enough free entries to processes later requests. This allows the throughput of an interleaved cache with a reorder buffer to approach that of a multi-ported design with reordering. When the queue size reaches sixteen entries, both the multi-ported and interleaved approach (with reordering logic) achieve an average throughput of three references per cycle. These results show that very little performance penalty is paid for switching from a true multi-ported design to an interleaved design when aggressive reordering is performed and queue sizes are sufficiently large.

6.1.3 Combining Memory Requests

In addition to reordering, the cache access rate can be increased by using a technique called *reference combining*. Combining is a technique concurrently developed by Wilson, Olokotun and Rosenblum [62] and Austin and Sohi [65], which attempts to combine references to the same cache line into a single request.

Combining focuses cache resources on areas in the design that can benefit from spatial locality and works as follows: Accessing a storage element in a conventional cache can be thought of as indexing into a two dimensional matrix using the line selector and line offset fields of the effective address. Combining incorporates additional logic in the request buffer (reorder buffer), along with limited cache line multi-porting, to improve access throughput [3]. However, combining logic applied to an interleaved approach enables multiple references to the same cache line to be translated into a single cache line request with multiple line offsets. This allows multiple references to the same line while requiring only a single multi-ported line be included in the implementation.

This can be accomplished by placing the cache line into a temporary storage buffer (associated with each cache bank) which is capable of supporting multiple line offset requests. Duplicating memory cells in a single line buffer per cache bank does not increase the circuit size significantly and if enough spatial locality exists in the application, the performance of a single ported line access with a multi-ported line offset calculation should approach the performance of a true multi-ported cache design. Furthermore, when combining is implemented in each bank of an interleaved cache, the optimal throughput can be increased significantly; in a four-way interleaved cache with two-ported combining logic on each bank, up to

[3]Recall that a multi-ported cache allows more than one request to be processed regardless of the relationship between the addresses involved.

eight references can be processed by the cache during each cycle.

For combining to be effective, spatial locality must exist in the applications. Figure 6.3 shows the same probabilities of bank allocation shown in Figure 6.1, but with the *same bank* component split into two sub-components, *same line* and a modified *diff line* entry. The *same line* components indicates the percent of consecutive references that access not only the same bank, but also the same line; these are amenable to combining. The new *diff line* component indicates the percent of consecutive references that access the same bank, but a different line. These reference cannot be combined, and must be processed sequentially due to the restrictions of the interleaved model. The spatial locality inherent in the applications is revealed in the figure; more than half of the consecutive references that map to the same bank also map to the same line. This means that improvement should be seen in an interleaved cache containing combining logic.

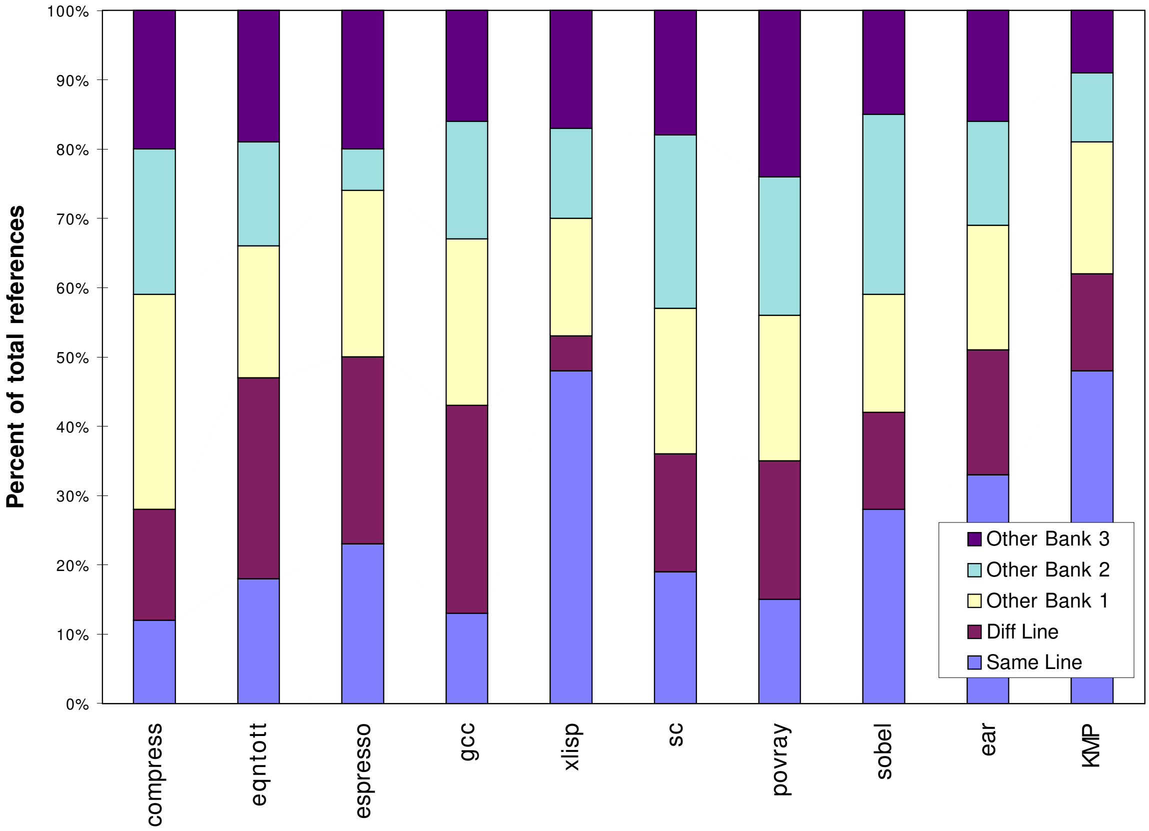

Figure 6.3: Relative distribution of cache line/bank references

The effects of combining on cache performance can be seen with and without reordering in Figure 6.4. Combining without reordering provides a small improvement over the interleaved cache with neither reordering nor combining. This improvement comes from the capture of those *same line* references that exhibit spatial locality. When the reorder buffer is included, combining does not need to be restricted to consecutive references, but can be applied to any reference in the re-order buffer. Very little additional complexity is required to perform combining throughout the re-order buffer — the comparison circuit already exists to determine whether a memory conflict occurs. Figure 6.4 shows the result of incorporating re-ordering and combining with an interleaved cache design. When a reorder buffer is included the throughput immediately increases, and then as buffer queue size increases throughput approaches that of a similarly configured multi-ported cache. Adding combining to the reorder buffer makes the performance curve of the inter-leaved cache design match that of a multi-ported design even for relatively small reorder buffers.

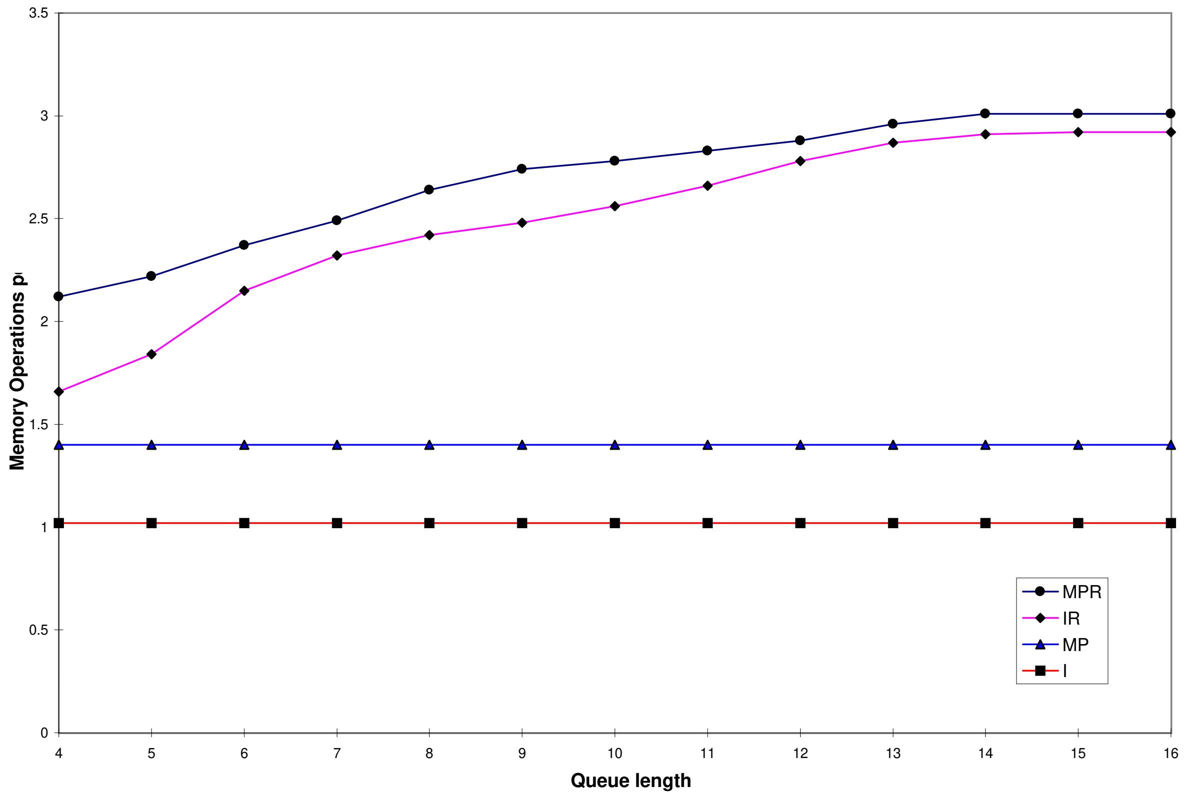

Figure 6.4: MISC Cache: Effects of Combining Requests

6.2 Summary

The MISC cache is a high throughput cache capable of supporting memory access requests from a set of cooperating processing elements. This design combines an efficient pipelined, interleaved cache with an aggressive reorder and combining buffer to achieve cache throughput performance approximating that of a high-performance multi-ported configuration.

This new cache structure combines the scalability of an interleaved cache with a mechanism to overcome the limitations of utilizing an interleaved approach. This study showed that when memory operations can be reordered, the performance of an interleaved design can approach that of a multi-ported design once the size of the buffer exceeds some threshold. Including combining logic enables multiple references to the same cache line to be merged into a single request, which has the affect of further improving the performance of the more restrictive interleaved approach. With the inclusion of a small reorder buffer, a simple interleaved cache design can match the performance of the more complex multi-ported cache. Both reorder buffers and combining logic can be incorporated into the cache design independent of the processor model; this design applies equally well to superscalar or VLIW architectures as it does to MISC.

Chapter 7

Conclusions

This research was originally motivated by the desire to examine the feasibility of exploiting instruction level parallelism in a scalable decoupled design. At the start of the project it was not clear whether a multiple instruction stream architecture would be capable of executing very tightly coupled applications without requiring complex architectural and compiler support. Previous approaches using a decoupled mechanism showed great promise in reducing the effects of high memory latency in a set of well structured scientific applications; however, general purpose, integer based applications were not studied. This dissertation explored the use of a decoupled processor design in executing a more general set of applications.

Contributions of this dissertation include:

- *Development of a scalable decoupled processor* capable of partitioning a task across multiple processing elements and exploiting instruction level parallelism. The results showed that near optimal performance can be achieved for the class of applications that were originally targeted by decoupled designs.

- *Development of a Compiler Model to Support the Architecture* to study the ability to partition more general applications across processing elements. This study showed that when modest levels of parallelism exist and can be exploited with a decoupled design, the unrestricted use of pointer variables severely limits the performance gains that can be made while guaranteeing program correctness. This study was limited by the unidirectional dependence flow designed into the compiler and the methods used to measure it. Further study of compilation techniques is warranted.

- *Development of a Unique Cache System to Improve Memory Performance.* The structure of the MISC cache incorporates several unique features to improve the ability of the cache to process a high rate of memory requests per cycle. A decoupled reorder buffer enables better utilization of simple interleaved cache design by reducing the effects of bank conflicts through a dynamic scheduling of memory requests.. Combining is also included to further improve the performance of the cache by exploiting cache line locality in a single multi-ported lines associated with each cache bank. These two features enable a simple interleaved cache design to achieve performance comparable to high performance multi-ported cache designs.

7.1 Future Directions

The research performed in this dissertation can be extended in a variety of ways. Improvements can be made to both the hardware and compiler aspects of the work discussed in the previous chapters. In this section, I propose modifications that can be made to the MISC design to better exploit instruction level parallelism. At the same time, capabilities inherent in the MISC design that can be incorporated into more conventional, single instruction stream architectures are discussed.

7.1.1 Hybrid Decoupled/Superscalar Design

The MISC architecture is capable of exploiting instruction level parallelism by separating the code into multiple, single-issue instruction streams; however, each of these instruction streams may contain additional ILP. An obvious extension to the MISC design is to implement each MISC processing element as a multiple issue processor. This allows the very tightly coupled execution characteristics found in superscalar design to be exploited, while at the same time allowing better scalability because of the more distributed resource allocation used in the decoupled design. Using this approach, for example, it is possible to construct a 16 issue architecture by incorporating four processing elements, each capable of issuing four instructions per cycle.

The partitioning results described in chapter 4 suggest that a non-homogeneous approach to designing the processing elements could potentially achieve even greater levels of ILP by matching the resources of each PE with the expected computational demand. The lead PE often executes the greatest number of instructions. This suggests that resources should be allocated to increasing the *scalarity* of the lead processor [1]. Future research should focus on the relationship between the partitioning strategy in the compiler and the resource partitioning among the processing elements; however, design time benefits may favor a homogeneous approach.

7.1.2 Compiler Development

The MISC compiler was implemented by modifying the vpcc compiler to support the code partitioning strategies (and other components) described in this disserta-

[1] The term *scalarity* has recently been adopted to identify the width of a superscalar design; a superscalar processor capable of executing up to three instructions per cycle is said to have a *scalarity* of three.

tion. The vpcc compiler was the best choice at the start of the project and provided adequate functionality to demonstrate the capabilities of the partitioning required to generate code for the MISC design. However, vpcc lacks certain features which would provide more optimal results. First, vpcc lacks the capability to perform the memory alias analysis necessary to achieve the improved partitioning shown in the Memory Partitioning strategy. Another deficiency (in terms of this research) is that vpcc is not in the public domain; this severely limits the ability to freely distribute the MISC compiler.

Since the initiation of the MISC compiler project, Stanford University has made a public release of the SUIF compiler. This compiler can be freely modified and re-distributed, requiring only the retention of the copyright notice. Furthermore, the SUIF compiler is designed to exploit both instruction level and data parallelism by performing sophisticated pointer analysis. These features make it a better base compiler for further development of the ideas presented in chapter 4. Changing compiler platforms is a difficult task for an architecture as complex as MISC, but necessary to continue the investigation of the limits to the MISC design and to enable the compiler to be released to other researchers.

7.1.3 Incorporating Register Queues in Superscalar Designs

One aspect of the MISC design that can be directly incorporated in standard architectures is the use of register queues. The use of register queues allow the compiler to schedule more registers than are available using the relatively few operand specification bits allocated to operand addressing, enabling the compiler to trade high register pressure for better instruction schedules.

In order to incorporate the ideas in this dissertation in existing designs it is necessary to make any modifications of the instruction set architecture as non-invasive as possible. Complete transparency is the obvious goal, but where that cannot be achieved, the simplicity of translating existing binaries is needed. Much the same problem was faced by Intel when it added floating point operations to the x86 line of microprocessors.

When the 8086 processor was originally developed to replace the 8080, floating point support was not included in the instruction set. As the processor gained in popularity, floating point capability had to be added to the ISA. Unfortunately there were not enough bits left in the format to support a conventional register operand addressing mode, so a mechanism had to be developed to specify the source and destination of an operation in fewer bits. This lead to the development of the *floating point register stack*. Most x86 floating point operations take the top two values off the register stack and place the result back on the stack. The successful inclusion of the floating point stack in the x86 demonstrates the circuit design does not pose significant implementation difficulties in the processor design. Fundamentally there is no difference between the design of a register file supporting stack access and one supporting queue access.

This demonstrates a way to incorporate a queuing discipline on top of a single register specifier (e.g. R1) without requiring drastic changes in the instruction set or the circuit design. For example, the implementation of the queue register semantics on an existing instruction set can be achieved without effecting previous code by overloading the semantics of an existing register and using a mode bit (as used in the real/segmented x86 address calculations) to activate the queue feature or by modifying the operating system to perform load time register reallocation for those existing binaries.

7.1.4 Prefetch Co-processor

Decoupled architectures offer a complementary approach to cache structures in reducing the effects of slow main memory. These architectures work well for applications with a well structured access pattern, whereas caches work well with applications that display locality of reference. Currently, researchers are exploring methods to achieve better cache performance by prefetching those items with little locality, but with a pattern of reference that can be captured by the insertion of specialized instructions by the compiler. This is exactly what decoupled designs do well. It is difficult to balance the load of instructions across multiple processing elements in applications with complex, intertwined data dependencies; this prevents a large speedup in execution. By changing the function of the access processor to that of a prefetch co-processor for a conventional superscalar architecture, the need to incorporate every dependence that affects memory access is removed. Because of the transparent nature of the cache, incorrect prefetches will result in slower (but still correct) program execution. Recent studies have shown that a variety of hardware feedback mechanisms can provide information allowing speculative prefetch hardware to improve overall cache performance [66]. The inclusion of an equally transparent prefetch co-processor decoupled from the execution of the main processor, yet with the assistance of hardware feedback, opens a new area of research.

This approach has a number of advantages. The transparent nature of the cache allows the inclusion of a co-processor to help in cache management regardless of the architectural specification of the system. Old programs can run without modification, yet statistics can be gathered and the prefetch co-processor code can be generated in order to improve performance. The decision on whether to incorporate the prefetch code can be made at any time for any set of applications. There is no need to modify the architecture of the existing processor to improve cache performance. Why make a modification which is visible to the processor architecture to improve performance of a cache unit that is supposed to be transparent? It is better to incorporate a transparent controller (a decoupled processor) to manage that resource.

7.1.5 Epilog

The MISC architecture presented in this dissertation demonstrates the feasibility of using a decentralized processor design to exploit a level of parallelism previously relegated to highly centralized processor designs. Furthermore, some features inherent in the MISC design can be easily incorporated into more conventional single-instruction stream approaches. This work suggests a number of new areas for research in finding scalable methods to exploit instruction level parallelism. The extensions to the MISC design discussed in the future work section highlight a few of these new research areas.

Bibliography

[1] Brian Moore, Andris Padegs, Ron Smith and Werner Buchholz, Concepts of the System/370 Vector Architecture, *Proceedings of the 14th Annual International Symposium on Computer Architecture*, (1987) 282-288.

[2] Mehrad Yasrebi and G. J. Lipovski, A State-of-the-Art SIMD Two-Dimensional FFT Array Processor, *Proceedings of the 11th Annual International Symposium on Computer Architecture*, (1984) 21-27.

[3] Toshio Kondom, Toshio Tsuchiya, Yoshihiro Kitamura, Yoshi Sugiyama, Takashi Kimura and Takayoshi Nakashima, Pseudo MIMD Array Processor — AAP2, *Proceedings of the 13th Annual International Symposium on Computer Architecture*, (1986) 330-337.

[4] Guang R. Gao, Lubomir Bic and Jean-Luc Gaudiot, Advanced Topics in Dataflow Computing and Multithreading, *IEEE Computer Society Press*, (1995).

[5] B. Ramakrishna Rau and Christopher D. Glaeser, Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing, *Proceedings of the 14th Annual Workshop on Microprogramming* (1981) 183-198.

[6] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Ken Mackenzie and Donald Yeung, The MIT Alewife Machine: Architecture and Performance, *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, (1995) 2-13.

[7] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta and John Hennessy, The DASH Prototype: Implementation and Performance, *Proceedings of the 19th Annual Symposium on Computer Architecture*, (1992) 92-103.

[8] Steven Reinhardt, Mark Hill, James Larus, Alvin Lebeck, James Lewis and David Wood, The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers, *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, (1993) 48-60.

[9] James E. Smith, Decoupled Access/Execute Computer Architectures, *Proceedings of the 9th Annual International Symposium on Computer Architecture*, (1982) 112-119.

[10] Matthew Farrens, Gary Tyson and Andrew Pleszkun, Study of Single-Chip Processor/Cache Organizations for Large Numbers of Transistors, *Proceedings of the 21st Annual International Symposium on Computer Architecture*, (1994) 338-347.

[11] Gary Tyson, Matthew Farrens and Andrew R. Pleszkun, MISC: A Multiple Instruction Stream Computer, *Proceedings of the 25th Annual Symposium and Workshop on Microprogramming and Microarchitectures*, (1992) 193-196.

[12] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth and Paul K. Rodman, A VLIW Architecture for a Trace Scheduling Compiler, *Proceedings of the Second International Conference on Architectural Support for Languages and Operating Systems*, (1987).

[13] B. R. Rau, D.W.L. Yen, W. Yen and R. A. Towle, The Cydra 5 departmental supercomputer: Design philosophies, decisions and tradeoffs, *Computer*, Volume 22 (1989) 12-34.

[14] Apollo Computers, The Series 10000 Personal Supercomputer: Inside a New Architecture, *Apollo Computers, Chelmsford, Mass.* (1988) 2-88.

[15] IBM, Special Issue on the IBM RISC System/R6000 processor. *IBM Journal Research and Development 34-1* (1990).

[16] Intel Corp., i860 64-Bit Macroprocessor Programmer's Reference Manual, Pub. No. 270710-001, *Intel Corp., Santa Clara, Calif* (1990).

[17] James E. Smith, Decoupled access/execute computer architectures, *ACM Transactions on Computer Systems*, (1984) volume 2, 289–308.

[18] B. Ramakrishna Rau and Joseph A. Fisher, Instruction-Level Parallel Processing: History, Overview and Perspective, *Journal of Supercomputing*, Volume 7-1(1993) 9-50.

[19] James. E. Thorton, Parallel Operation in the Control Data 6600, *AFIPS Proceedings of the Spring Joint Computer Conference, part II*, 26 (1964) 33-40.

[20] John Hennessy and David Patterson, Computer Architecture: A Quantitative Approach, *Morgan Kaufmann, San Mateo, California*, (1990).

[21] R. M. Tomasulo, An Efficient Algorithm for Exploiting Multiple Arithmetic Units, *IBM Journal* 11 (1967) 25-33.

[22] Wm. Wulf, Evaluation of the WM Architecture, *Proceedings of the 19th Annual Symposium on Computer Architecture*, (1992) 382-390.

[23] Richard L. Sites, Alpha AXP Architecture, *Communications of the ACM*, Volume 36-2 (1993) 33-44.

[24] J. R. Goodman, J. T. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter and H. C. Young, PIPE: a VLSI Decoupled Architecture, *Proceedings of the 12th Annual International Symposium on Computer Architecture*, Volume 2 (1985) 20-27.

[25] G. L. Craig, J. R. Goodman, R. H. Katz, A. R. Pleszkun, K. Ramachandran, J. Sayah and J. E. Smith, PIPE: A High Performance VLSI Processor Implementation, *Journal of VLSI and Computer Systems*, (1987).

[26] Honesty C. Young and James R. Goodman, A Simulation Study of Architectural Data Queues and Prepare-to-Branch Instruction, *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, (1984) 544-549.

[27] Matthew Farrens and Andrew Pleszkun, Implementation of the PIPE Processor, *Computer* (1991) 65-70.

[28] Honesty Cheng Young, Evaluation of a Decoupled Computer Architecture and the Design of a Vector Extension, *Ph.D Thesis, University of Wisconsin-Madison*, (1985).

[29] R. Gupta, A Fine-grained MIMD Architecture based upon Register Channels, *Proceedings of the 23rd Annual Symposium and Workshop on Microprogramming and Microarchitectures*, (1990) 54-64.

[30] A. V. Aho, R. Sethi and J. D. Ullman, Compilers Principles, Techniques and Tools, *Addison-Wesley Publishing*, (1986).

[31] R. Gupta, The Fuzzy Barrier: A Mechanism for High-speed Synchronization of Processors, *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems* (1989).

[32] Carlo H. Sequin and David A. Patterson, Design and Implementation of RISC1 University of California, Berkeley Technical Report CSD-82-106 (1982).

[33] G. S. Tjaden and M. J. Flynn, Detection and Parallel Execution of Independent Instructions, *IEEE Transactions on Computer*, Volume 19-10 (1970) 889-895.

[34] Norman P. Jouppi and David W. Wall, Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines, *Proceedings of the Third International Conference on Architectural Support for Languages and Operating Systems*, (1989) 272-282.

[35] D. J. Kuck, Y. Muraoka, and S. C. Chen, On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up, *IEEE-TC*, (1972) Vol C-21, 1293-1310.

[36] Todd Austin and Gurindar Sohi, Dynamic Dependency Analysis of Ordinary Programs, *Proceedings of the 19th Annual International Symposium on Computer Architecture*, (1992) 342-351.

[37] Michael Butler, Tse-Yu Yeh and Yale Patt, Single Instruction Stream Parallelism is Greater than Two, *Proceedings of the 18th Annual International Symposium on Computer Architecture*, (1991), 276-286.

[38] M. S. Lam and R. P. Wilson, Limits of control flow on parallelism, *Proceedings of the 19th Annual International Symposium on Computer Architecture*, (1992) 46-58.

[39] M. E. Benitez and J. W. Davidson, Code Generation for Streaming: an Access/Execute Mechanism, *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, (1991) 132-141.

[40] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank and R. A. Bringmann, Effective Compiler Support for Predicated Execution Using

the Hyperblock, *Proceedings of the 25th Annual International Symposium on Microarchitecture*, (1992) 45-54.

[41] M. S. Lam, Software Pipelining: An Effective Scheduling Technique for VLIW Machines, *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, (1988) 318-328.

[42] SPEC CINT92 and CFLOAT92, Release V1.1, (1992)

[43] SPEC CINT89 and CFLOAT89, Release V1.0, (1989)

[44] POVRAY team, Persistence of Vision Ray Tracer (POV-Ray) User's Documentation, Version 2.0, *Unpublished Document*, (1993).

[45] Anil K. Jain, Fundamentals of Digital Image Processing, *Prentice-Hall, Englewood Cliffs, N.J.*, ISBN 0-13-332578-4, (1989).

[46] Thomas H. Cormen, Charles E. Leiserson and Ronald L. Rivest, Introduction to Algorithms, *The MIT Press*, ISBN 0-262-03141-8, (1990) 869-876.

[47] F. H. McMahon, LLNL FORTRAN KERNELS: MFLOPS, *Lawrence Livermore Laboratories*, (1984).

[48] Thomas H. Cormen, Charles E Leiserson and Ronald L Rivest, Introduction to Algorithms, *McGraw-Hill Book Company*, ISBN 0-07-0013143-0 (1990).

[49] Gary Scott Tyson, The Effects of Predicated Execution on Branch Prediction, *Proceedings of the 27th Annual International Symposium on Microarchitecture*, (1994) 196-206.

[50] M. Johnson, Superscalar Processor Design, *Prentice-Hall, Englewood Cliffs, N.J.*, (1991).

[51] Matthew K. Farrens and Andrew R. Pleszkun, Strategies for Achieving Improved Processor Throughput, *Proceedings of the 18th Annual International Symposium on Computer Architecture*, (1991) 362-369.

[52] J. R. Goodman, J. T. Hsieh, K. Liou, A. R. Pleszkun, P. B. Schechter and H. C. Young PIPE: a VLSI Decoupled Architecture, *Proceedings of the 12th Annual International Symposium on Computer Architecture*, (1985) 20-27.

[53] M. S. Lam, The SUIF Interface Format, *Stanford University*, (1995).

[54] J. E. Smith, S. Weiss and Nicholas Y. Pang, A Simulation Study of Decoupled Architecture Computers, *IEEE Transactions on Computers*, Volume C-35-8 (1986) 692-702.

[55] Matthew Farrens, Phil Nico and Pius Ng, A Comparison of Superscalar and Decoupled Access/Execute Architectures, *Proceedings of the 26th Annual International Symposium on Microarchitecture*, (1993) 100-103.

[56] D. Bernstein, D. Cohen, Y. Lavon and V. Rainish, Performance Evaluation of Instruction Scheduling on the IBM RISC System/6000, *Proceedings of the 25th Annual International Symposium on Microarchitecture*, (1992) 226-235.

[57] B. R. Rau, M. Lee, P. P. Tirumalai and M. S. Schlansker, Register Allocation for Software Pipelined Loops, *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, (1992) 283-299.

[58] Digital Equipment Corporation, Maynard, Mass. DECchip 21064 Microprocessor: Hardware Reference Manual, (1992).

[59] Amitabh Srivastava and Alan Eustace, ATOM: A system for building customized program analysis tools, *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, (1994) 196-205.

[60] Digital Equipment Corporation, Maynard, Mass. DECchip 21164 Microprocessor: Hardware Reference Manual.

[61] Manoj Franklin and Gurindar Sohi, Register Traffic Analysis for Streamlining Inter-Operation Communication in Fine-Grain Parallel Processors, *Proceedings of the 25th Annual International Symposium on Microarchitecture*, (1992) 236-245.

[62] Kenneth Wilson, Kunle Olokotun and Mendel Rosenblum, Increasing Cache Port Efficiency for Dynamic Superscalar Microprocessors, *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, (1996) 147-157.

[63] Motorola, The R10000 Reference Manual, (1996).

[64] Keith I. Farkas and Norman P. Jouppi, Complexity/Performance Tradeoffs with Non-Blocking Loads, *Proceedings of the 21rd Annual International Symposium on Computer Architecture*, (1994) 211-222.

[65] Todd Austin and Gurindar Sohi, High-Bandwidth Address Translation for Multiple-Issue Processors, *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, (1996) 158-167.

[66] Tien-Fu Chen and Jean-Loup Baer, A Performance Study of Software and Hardware Data Prefetching Schemes, *Proceedings of the 21rd Annual International Symposium on Computer Architecture*, (1994) 223-232.

Appendix A

MISC Instruction Set

Appendix A has been removed in this condensed version of the dissertation. MISC instructions fall into seven categories:

```
Scalar Instructions:
add     and     fadd    fmul    fsub    mul
or      sll     sra     srl     sub     xor

Compare and Branch Instructions:
ba      bnz     bz      ceq     cge
cgt     fceq    fcge    fcgt

Predicate Instructions:
addp    andp    faddp   fmulp   fsubp   mulp
orp     sllp    srap    srlp    subp    xorp

Vector Instructions:
addv    andv    faddv   fmulv   fsubv   mulv
orv     sllv    srav    srlv    subv    vloop
xorv

Sentinel Instructions:
adds    ands    fadds   fmuls   fsubs   muls
ors     slls    sloop   sras    srls    subs
xors

Memory Instructions:
laq     laq2    saq     laqv    laq2v   saqv

Special Instructions:
cvtif   cvtfi   qempty  qoff    qon
```