Performance Evaluation of the PIPE Computer Architecture

by

Jian-tu Hsieh
Andrew R. Pleszkun
James R. Goodman

Computer Sciences Technical Report #566

November 1984

Performance Evaluation of the PIPE Computer Architecture

Jian-tu Hsieh, Andrew R. Pleszkun and James R. Goodman

Computer Sciences Department
University of Wisconsin
1210 W. Dayton St.
Madison, WI 53706

ABSTRACT

PIPE (Parallel Instruction and Pipelined Execution) is a high performance computer architecture with several features that make it well-suited for VLSI implementation. A PIPE machine consists of a memory controller and two co-processors. An execute processor computes operand addresses for the execute processor that performs the main calculations of a program running on the PIPE system. To study PIPE's performance, we implemented a simulator through which we ran a set of hand-coded benchmarks. The dependence of performance on both program and hardware characteristics is investigated. Various parameters and execution modes of the PIPE architecture are considered. Simulation results demonstrate the usefulness of architectural queues and decoupled architectures. Simulation results also show that the PIPE architecture performs very well on the benchmark programs.

Table of Contents

1. Introduction

PIPE (Parallel Instruction and Pipelined Execution) is a decoupled computer architecture with several features that make it well suited for VLSI implementation. PIPE was first proposed in [SPKG83] as a research vehicle for studying high performance VLSI architecture and organization. Since then, a great amount of effort has been devoted to the study of various aspects of this architecture. The PIPE processor is intended for state-of-the-art single-chip VLSI fabrication. The design of a prototype PIPE processor was described in [CGKP83]. Several major portions of the processor have been laid out in nMOS technology and submitted for fabrication and many software tools have been developed for studying the PIPE architecture.

Very Large Scale Integration (VLSI) is the current trend in computer technology [MeCo80]. A very powerful single-chip processor is a good candidate for use of the large number of electronic devices provided on a silicon chip [PaSe80]. Well-established pipelining design philosophy [Kogg81, RaLi77] and VLSI technology are a good match. A pipelined computer organization requires more logic than a serial one, but it does not necessarily require more interface pins. The additional logic required for pipelining contributes significantly to system performance by allowing parallel execution in the pipeline stages.

High performance machine design should exploit parallelism at all system levels, both within processors and among processors. PIPE is a decoupled architecture that allows parallel execution of a single task in two processors. A decoupled architecture provides a clean partition of computing functionals so that they can be implemented in two cooperating processors. Communication among the processors and the main memory is buffered by architectural queues that smooth out temporary congestion in busses and allow "elastic" coupling between the two processors.

To validate the features of the PIPE architecture, we decided to perform detailed simulation of the system. For such a study, a functional interpreter and performance simulator were implemented. In order to justify the design rationale and to identify possible weaknesses of the PIPE architecture, we decided to evaluate the performance of the PIPE system in a particular application environment.

To this end, we used the Lawrence Livermore Laboratories loops [McMa72] as benchmark programs.

The rest of the paper is organized as follows. In the next section, we give more details of the PIPE architecture. Section 3 describes the performance simulator and some implementation details of a PIPE system. As benchmarks, we used hand-coded Lawrence Livermore loops and these are discussed in Section 4. Simulation results of the Lawrence Livermore loops are presented in Section 5. We also analyze the results and discuss some programming techniques and performance/hardware trade-off. Section 6 contains a conclusion.

2. The PIPE Architecture

The PIPE (Parallel Instruction and Pipelined Execution) architecture is intended for very-high-performance VLSI computers. The architecture, the motivation behind it, and some of the features of its planned implementation are discussed in [SPKG83]. More details on implementing a prototype PIPE processor are described in [CGKP83]. In this section, we briefly describe the important features of PIPE and some related architectures. More implementation details will be discussed in a later section, in conjunction with the PIPE simulator.

Some known impediments to computer performance have been taken into consideration in the design of the PIPE architecture. In a conventional von Neumann architecture, the CPU interacts with one memory over one bus. Both instructions and operands are transferred via this bus. In the VLSI environment, bus width is usually limited to the processor's word size. The speed of bus communications is limited by the power-delay product [MeCo80] of the circuit. The von Neumann bottleneck is the limitation on the bus traffic posed by the bus bandwidth. An on-chip instruction cache is implemented for each PIPE processor to reduce the number of words passed through this bottleneck [Good83]. Flynn [Flyn66] observed that in the instruction fetch/decode path there is some bottleneck through which instructions pass at the maximum rate of one per clock period. Decoupled architectures, with their two parallel instruction streams, reduce the constraint of the Flynn bottleneck.

PIPE is a Register-to-Register (RR) architecture in which all the operands of arithmetic and logical instructions come from registers or queues. Similarly, the operation results go to registers or queues. *Load* and *Store* are the only instructions that access main memory. Since the resource and time requirements of register-to-register operations are known in advance, RR architectures are better suited for pipelining than memory-oriented architectures.

Some processors have simple instruction sets. The argument for these processors is that they are easy to design and implement, and can execute instructions quickly. For example, the Reduced Instruction Set Computer, RISC-I [PaSe81], has a 400ns cycle time and can execute one instruction per cycle. In the pipelined implementation of a PIPE processor, an instruction may take several clock periods to finish, but we can issue one instruction per clock period. A PIPE processor has an *elemental* instruction set similar to that of RISC-I. An elemental instruction is one whose resource requirements can be readily determined before instruction execution. Pipeline conflict detection and interlocks can be done in a single instruction issue stage that precedes the execution pipeline. The instruction issue stage for an elemental instruction set can be implemented with relatively simple logic. Using nMOS technology, a PIPE processor is expected to have a basic clock period which is about an order of magnitude shorter than the RISC-I cycle time.

The main feature of decoupled architectures is a high degree of decoupling between operand access and execution. This results in an implementation which has two separate instruction streams and processors that communicate via queues. The MAP-200 array processor [CoSt81] proposed the first decoupled architecture in the literature. The MAP-200 has an integer addresser separate from its floating-point arithmetic unit. The two units run independently except that some synchronization is needed at certain points, such as the end of a loop. Software is called on to do most of the coordination and synchronization between these two units. Another decoupled architecture based on CRAY-1 computers [Russ78] was proposed in [Smit82]. It tried to remove the burden of synchronization from the programmers. This architecture uses queues to pass branch decisions between the cooperating processors and eliminates some unnecessary waiting periods suffered in the MAP-200

architecture. The Structured Memory Access (SMA) architecture [Ples82, PlDa83] also falls in the category of decoupled architectures. It uses a computation processor and a decoupled memory access processor with special hardware for efficient accessing of program and data structures and for effective branch and loop management.

PIPE is similar to the decoupled architectures described above, but with special considerations for pipelined VLSI implementation. Although a single PIPE processor can execute a program by itself, we are more interested in the decoupled configuration where two processors execute a single process divided into two parallel instruction streams. The Access Processor (AP) computes operand addresses and makes memory references for the Execute Processor (EP) that performs the main calculations of a program. Each processor has an on-chip instruction cache that reduces the effect of the von Neumann bottleneck. Both the instruction fetch unit and the Arithmetic Logical Unit (ALU) are pipelined. The Prepare to Branch (PBR) instructions are designed to facilitate smooth transfer of control. A PBR instruction gives the instruction fetch logic advance notice of a branch and minimizes its interruption of the instruction unit pipeline.

A memory control unit serves both processors and is responsible for detection and resolution of memory hazards. All memory transactions in a PIPE system are buffered by hardware queues. These queues allow memory access time to be overlapped with useful processing. The branch queues between the two processors provide elastic coupling between the two processors' execution. Thus a decoupled architecture automatically performs some code scheduling between the two processors at run time. The intent of a decoupled architecture is that the access processor should run ahead of the execute processor and reduce or eliminate observed memory delay.

3. The Simulation Tools

Many software tools have been developed for studying the PIPE architecture. Among them are a Pascal compiler, a code scheduler, an assembler, a functional interpreter, and a performance simulator. All these tools work for both single-processor and decoupled access/execute modes.

The interpreter takes PIPE object code as input, interprets it on an instruction-by-instruction basis and produces both text and binary trace files. The binary trace files contain the following information for performance simulation:

(1) addresses of data reads,
(2) addresses of data writes,
(3) addresses of instruction reads,
(4) the instructions.

A very versatile cache simulator has been developed that uses this binary trace format and has been used extensively in studying the cache organizations with PDP-11 and VAX traces [Good83, SmGo83]. This simulator can be modified slightly to add hit/miss tags to the PIPE instructions in the binary trace files. Currently, however, the PIPE simulator only has a simple built-in instruction cache simulator that simulates a direct-mapped cache.

The PIPE simulator takes binary trace files as input, simulates the execution of a PIPE program on a clock-by-clock basis, and finally prints the statistics gathered during the simulation. The remainder of this section discusses the simulator in more detail. Detailed knowledge of the simulator, its operation, and its implicit assumptions is necessary for proper interpretation of the results in Section 5. Section 3.1 introduces the simulation language we used to implement the PIPE performance simulator. We use the model illustrated in Figure 3.1 to simulate a PIPE computer system. The system consists of two cooperating processors and a memory subsystem which services both processors. Section 3.2 discusses the simulation model for a PIPE processor. Section 3.3 describes a simple model for the memory system.

3.1. The Simulation Language Simpas

The PIPE simulator is written in the simulation language Simpas, version 5.0 [Brya83]. Simpas is an event-oriented simulation language embedded in Pascal. It has many attractive features.

First of all, it is extremely portable. It is implemented as a preprocessor that accepts an extended version of Pascal as input and produces a standard Pascal program as output. The preprocessor itself is written in standard Pascal, and the language has been designed so that it depends only

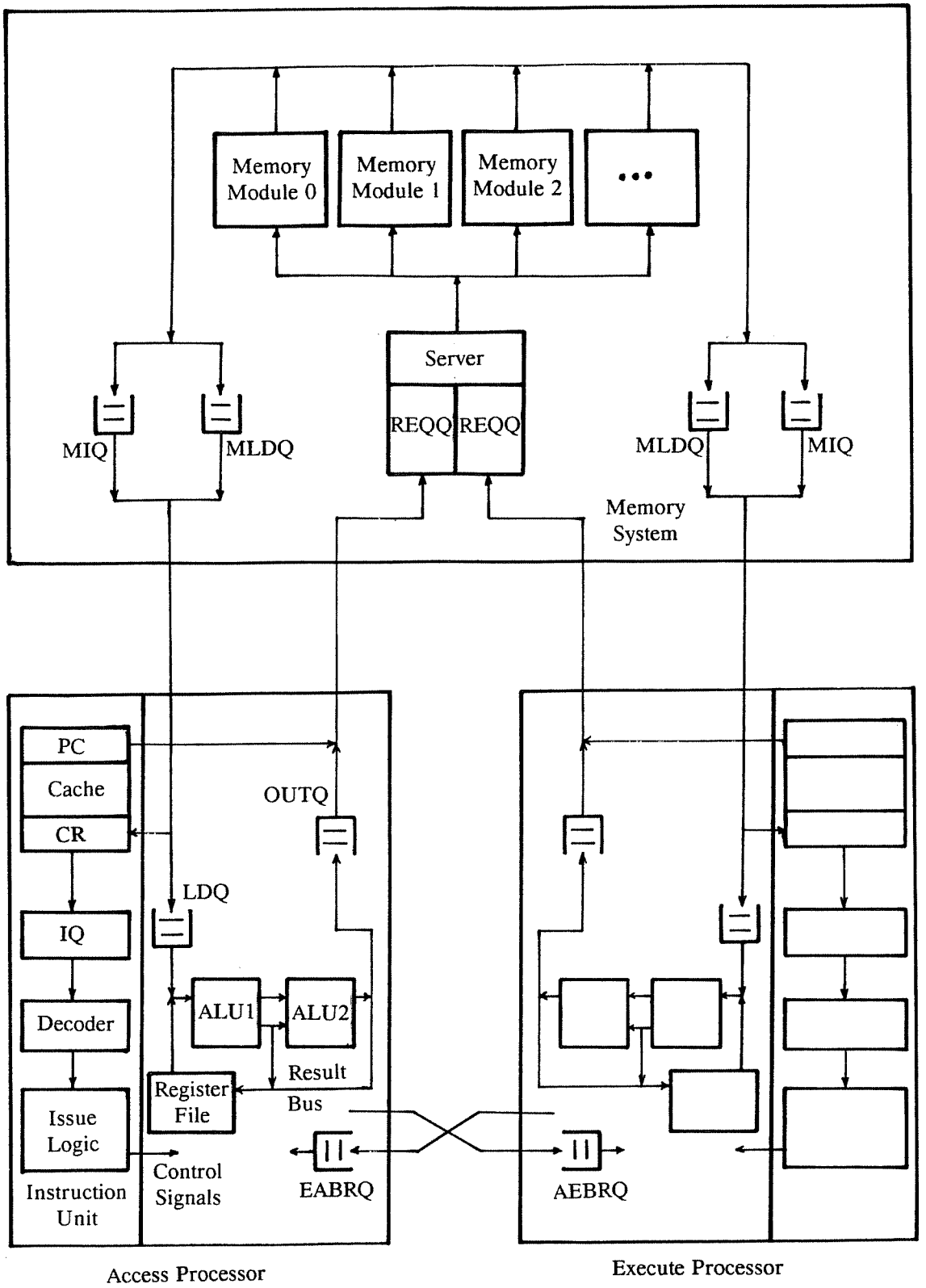


Figure 3.1. The block diagram for a **PIPE** machine.

on standard Pascal features.

Secondly, the language supports queues and operations on queues. This makes the simulation of the PIPE architectural queues very easy. The language also supports some error-detection at run-time. For example, it detects the error of having an element in two queues at the same time.

Finally, Simpas has automatic mechanisms for statistics collection. In order to gather statistics about an interested variable, a programmer simply specifies it to be of *watched* type. The preprocessor will generate appropriate code to collect statistics for watched variables. Since the change from a regular variable to watched variable, and vice versa, is very easy, we can make such changes during the development of a simulator depending on the execution efficiency and the amount of statistics needed.

The simulator used for this study was implemented in about 2,000 lines, excluding comments, of Simpas code. The simulator executes on a VAX-11/750 at the speed of about 30 ms user and system CPU time per PIPE clock cycle in decouple-mode simulation.

3.2. The PIPE Processors

The processor functions that we simulate are based on the proposed initial implementation of the PIPE processors [CGKP83]. The access processor and the execute processor can be identical: they are in fact simulated by the same procedure. However, we do not preclude the possibility of using two different processors. The simulator maintains two separate sets of simulation parameters for the processors and allows a user to specify each parameter independent of the others.

3.2.1. The Instruction Unit

A PIPE instruction consists of either one or two 16-bit parcels. The instruction unit is responsible for supplying an instruction stream for the ALU to execute. In order to reduce the number of accesses it makes to main memory, the instruction unit contains an on-chip instruction cache. Upon a cache hit, a line of instructions is loaded from the cache into the Cache Register (CR). Upon a cache miss, the miss address is sent to the memory control unit and the returned instructions are

assembled into a line in the CR. The instructions are processed through a pipeline consisting of the CR, the Instruction Queue (IQ), the decoder and the issue logic.

The branch instructions in the PIPE architecture are designed to minimize their interruption of the instruction unit pipeline. Branch target addresses are loaded into branch registers by separate instructions. A 'Prepare to Branch' (PBR) instruction specifies: (1) a condition to test, (2) a branch register, and (3) a 3-bit branch count. The branch count ranges from 0 to 7 and specifies the number of instruction parcels to execute before the branch is taken. Well-written programs keep the branch counts large, allowing the processor to gracefully change control flow.

In order to supply a smooth instruction stream, the instruction unit prefetches a line of instructions into the CR if the line will definitely be used. The decision is based on the early detection and partial decoding of PBR instructions. When the outcome of a branch condition is not yet known, a cache miss beyond the branch count will not be processed.

The bus width in VLSI processors is limited by the number of pins available on a package. The speed of bus communications limits the clock of the on-chip logic. The bus bandwidth also generates a potential bottleneck in the PIPE system. The PIPE architecture's prefetching strategy and small cache line size are designed to avoid unnecessary memory accesses.

3.2.2. The Execution Unit

The simple instruction set [PaSe81] of PIPE allows a well-structured pipelined implementation of the Arithmetic Logic Unit (ALU). The simulator assumes that the ALU is a two-stage pipeline. The first stage takes operands from the register file, the Load Data Queue (LDQ), or the instruction unit (immediate operands). Both stages can send data through the result bus to the register file or the Output Queue (OUTQ).

The Load Data Queue (LDQ) is the most important architectural queue in the PIPE architecture. It serves as a First-in First-out buffer for the data loaded by the *Load* instructions. The size of the LDQ is a simulation parameter. Ideally we like the queue to be infinitely long. So a large

number of *Load* instructions can be scheduled ahead of the instructions that use the data. The effect of the LDQ size on the system performance will be studied in later sections.

3.2.3. The Busses

Each processor communicates with the Memory Control Unit (MCU) through two dedicated busses, one for input and one for output. Bus communications are buffered by the queues on both sides of the busses. Tags are associated with all the bus traffic and designate the kind of information being transferred.

The MCU-to-CPU tags are:

(1) data: A data word loaded by a *Load* instruction has returned from the MCU. It should go to the LDQ.

(2) instruction: An instruction requested by a cache miss has returned. It should go to the Cache Register in the instruction unit.

(3) control: A control message is sent through the bus. One possible control message is that the bus is idle. Other kinds of control messages are used to coordinate the MCU-CPU communications and will be discussed shortly.

Each processor conceptually contains the following queues:

LAQ: load address queue,
ALAQ: alternate load address queue,
SAQ: store address queue,
ASAQ: alternate store address queue,
SDQ: store data queue.

Physically all these queues are combined into a single queue called the Output Queue (OUTQ). Cache-miss instruction addresses can be thought of as going through an Instruction Address Queue (IAQ). These queues are illustrated in Figure 3.2.

The data or addresses in the OUTQ are sent to the memory control unit in a strict first-in first-out order. In the event of a cache miss, the instruction address is sent through the same CPU-to-MCU bus. By default the simulator gives priority to the instruction address. The simulator

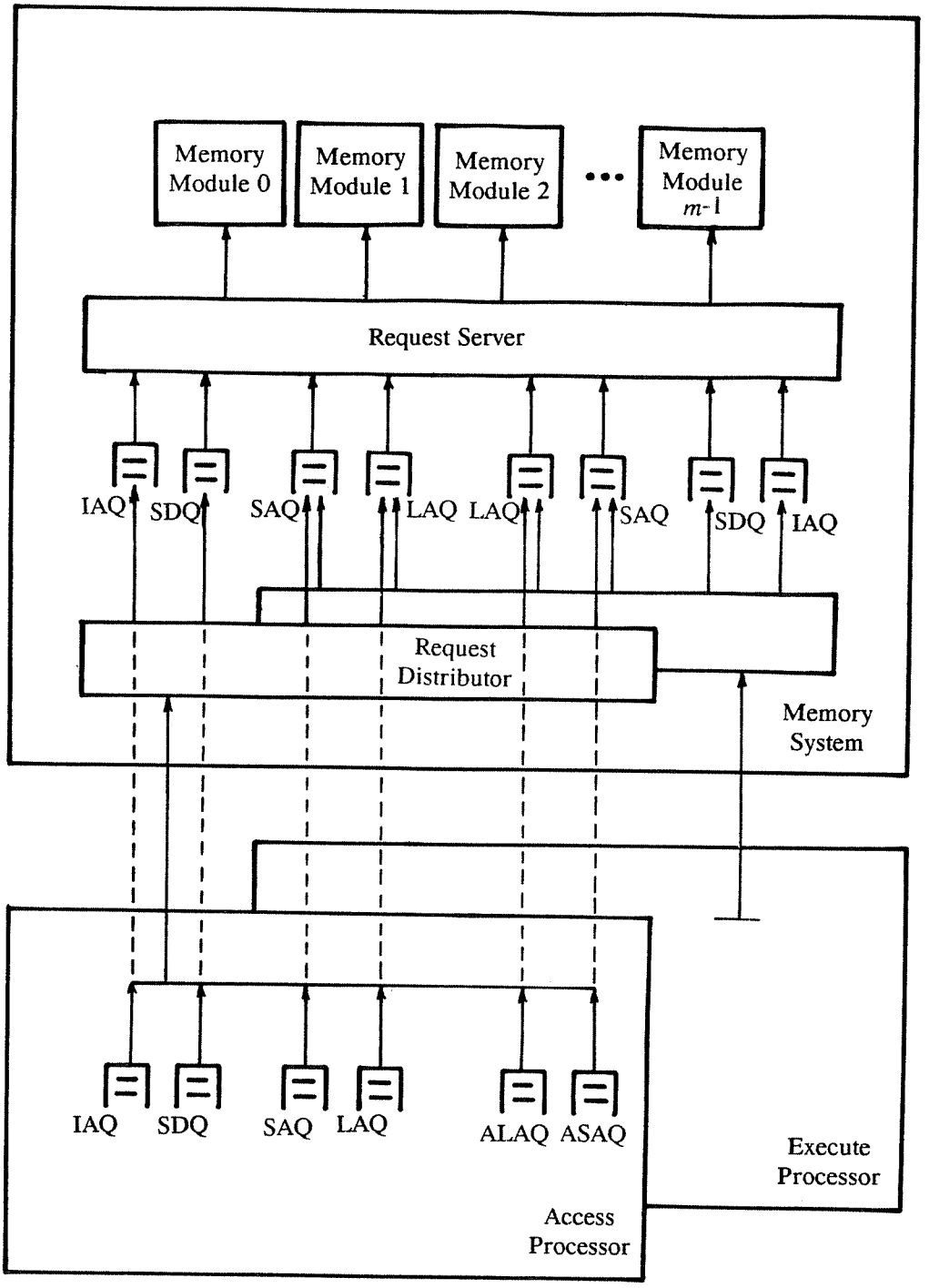


Figure 3.2. Memory request queues in a PIPE machine.

provides an option that deprives the instruction address of its priority and forces it to enter the OUTQ and wait for transmission to the MCU.

All the CPU-to-MCU bus traffic is also tagged with a source queue indicator. Another tag on the bus is for control messages. When the LDQ of a processor is full, the processor has to inform the MCU so that no more data will be sent to the LDQ. Similarly, a processor has to inform the MCU when the LDQ is not full and can be refilled. The *LDQ-refill* and *LDQ-full* messages can be considered as the set and reset signals to a flip-flop in the MCU. The MCU sends LDQ data only if the flip-flop is set. The *LDQ-full* message is the most urgent signal. In order to avoid any loss of data, it is given the highest priority to use the CPU-to-MCU bus. Usually the *LDQ-refill* message is not urgent and can be delayed if necessary. The *LDQ-low* parameter of the simulator is used to determine when to send the *LDQ-refill* message. If the LDQ length is less than *LDQ-low* then the *LDQ-refill* gets priority. Otherwise the *LDQ-refill* message is postponed unless the CPU-to-MCU bus is idle and can be used freely for this message.

A similar mechanism is used to control the MCU-to-CPU bus transmission. The control messages sent by the memory control unit are named *REQQ-full* and *REQQ-refill*.

3.2.4. The Issue of Instructions

PIPE and the CRAY-1 have similar philosophies about instruction issue: resources needed by an instruction are reserved at issue time [Cray79]. If an instruction to be issued would conflict with an already issued instruction in the execution pipeline, the instruction is blocked until the conflict is removed. In the PIPE architecture, possible reasons for blocking the instruction issue are:

(1) instruction not available,
(2) LDQ empty,
(3) OUTQ full,
(4) result bus contention,
(5) output BRQ full, and
(6) input BRQ empty.

The last two reasons are only for decoupled-mode executions where one processor has to wait for the other at some synchronization point. The simulator keeps a record of all these blockings. Such

information is very useful in identifying possible bottlenecks in a PIPE program or the PIPE machine.

3.3. The Memory Subsystem

The memory subsystem consists of the Memory Control Unit (MCU) and several memory modules. Design of a memory subsystem for PIPE and the study of various memory service strategies are in progress [Liou84]. A simple one-by-one memory service strategy is implemented as an integral part of the PIPE performance simulator.

3.3.1. The Memory Modules

PIPE's memory modules are interleaved for high performance: low order interleaving is used. If there are m memory modules then the memory word at address n resides in module n mod m. In real memory systems the number m is usually a power of two so that the least significant bits of an address determines which module the memory word resides in.

The simulator allows any number of memory modules in the memory system. If the number of memory modules is specified to be zero then the simulator ignores memory module conflicts and treats all the memory requests as pure delay of the amount specified by the memory access time unless the request is blocked by hazards (to be described below). This feature can be used to study the effect of memory module conflicts.

3.3.2. The One-by-One Service Strategy

Figure 3.1 shows that, in the memory system, all the requests are buffered in the Request Queues (REQQs) before they are serviced. In fact, the requests cannot be serviced in a strictly first-in first-out order. For example, an unmatched store address should not block the subsequent load requests. Figure 3.2 shows one way to avoid such blockings. As illustrated, the incoming bus traffic is distributed into various buffer queues according to the associated tags. An alternate load/store address from one processor is considered as a load/store address for the other processor. The dashed lines in the figure show the virtual paths between the pairs of queues. All the virtual

paths are concentrated into a single physical bus (and therefore necessitating the use of tags).

An address from the Store Address Queue (SAQ) and a data word from the Store Data Queue (SDQ) are matched by the server and are serviced as a single write operation. In order to detect possible Read-After-Write (RAW), Write-After-Read (WAR) or Write-After-Write (WAW) hazards, each request is time-stamped when it enters the memory control unit. The time stamp for a write operation is that of the SAQ element. In addition to the RAW, WAR and WAW hazards, there are also MLDQ hazards which will be discussed shortly.

Three general rules are observed in the memory system.

(1) Unmatched SAQ or SDQ elements must wait for their mates.

(2) Any request whose intended memory module is busy must wait.

(3) Any request that can cause hazards must wait.

Two more rules are enforced by the one-by-one memory service strategy.

(4) The requests in each separate queue are serviced in a strict first-in first-out order.

(5) If two or more requests are ready to be serviced then the one with earlier time stamp has priority.

An exception to the last rule is that the simulator gives instruction requests higher priority by default. The priority can be canceled optionally. The fourth rule insures that the data are returned to the processors in their original requested order. If some sequencing mechanism can guarantee the same effect then this rule can be lifted. The return order is important for instructions and the LDQ elements. The benefit of issuing requests out of order is studied in [Liou84].

The data read from the memory modules are buffered in the Memory Load Data Queue (MLDQ) or the Memory Instruction Queue (MIQ) and are sent to the processor through the bus. The simulator gives priority to instructions. Since there is at most one outstanding cache miss, the length of the MIQ is bounded by the cache line size. The MLDQ is essentially an extension of the LDQ. Since the MLDQ is of finite size, an MLDQ-overflow hazard will occur if a load request tries

to send data to a full MLDQ. The third rule above prevents such a request from being issued.

The simulator keeps track of the number of outstanding memory requests from each processor. A control message is sent to the processor when certain limits are reached. The limits for such *REQQ-full* and *REQQ-refill* control messages are simulation parameters.

4. The Benchmarks

The Lawrence Livermore Laboratories (LLL) loops [McMa72] are often used to measure the performance of computer systems. They are CPU-limited numerical computations extracted from scientific application programs and involve floating-point operations on arrays of one or more dimensions. Since the floating-point operations and data format are not fully defined in the PIPE architecture, in this study we treat all variables as integers. Under this restriction the study of some of the loops is not practical because they involve conversions between floating-point numbers and integers. In this study we hand-coded the first twelve loops and use the PIPE simulator to simulate their execution.

The LLL loops are originally coded in FORTRAN as DO loops. We translated the loops into Pascal and used the PIPE Pascal compiler and code scheduler to generate PIPE code. The translation was straightforward but the compiled code was inefficient. The simulation results shown in Table 4.1 can be used to compare hand-coded and compiled code. The execution times are measured in terms of the basic clock cycle of a PIPE machine. The last column shows the ratios between the execution times for the single-processor mode and the decoupled access/execute mode. The compiler generates inefficient code for accessing array elements. This is indicated by the results for loop 8, which operates on many three-dimensional arrays. Efforts to write an optimizing PIPE Pascal compiler are underway [Youn84]. (An improved PIPE Pascal compiler is available at the time of this writing.)

We feel that a performance study based on inefficient code may not be fruitful and its results may be misleading. Therefore, we hand-coded the first twelve LLL loops. The following rules guided our coding work for the single-processor mode of execution.

Program	SP		AE				Speedup
	#inst	exec time	AP #inst	EP #inst	total #inst	exec time	
LLL1 hand-written code	4009	4069	2406	2005	4411	2859	1.423
LLL1 compiled code	13609	18125	14409	2804	17213	16138	1.123
LLL2 hand-written code	8005	11053	5604	3603	9207	6851	1.613
LLL2 compiled code	14809	20422	20809	2804	23613	27882	0.732
LLL8 hand-written code	3330	3564	1412	1726	3138	1894	1.882
LLL8 compiled code	14115	16815	19994	1653	21647	32643	0.515

Table 4.1. Comparison of hand-written and compiled code for some LLL loops.

(1) The original loop structure is maintained. A Prepare to Branch (PBR) instruction at the end of the loop tests the termination condition and branches to the head of the loop when appropriate. We tried to maximize the branch count in the PBR instruction.

(2) As long as free registers are available, the loop-invariant quantities are kept in registers. These registers are loaded before the loop is entered. Similarly the branch target address is loaded into a branch register only once before the loop is entered.

(3) All *Load* instructions appear at the beginning of the loop in order to hide the memory access delay. However, this loading strategy may fill up the queues and cause deadlock in the single-processor mode, especially if the queues are short. In some loops it is necessary to intersperse the load instructions in the loop body to avoid deadlock.

(4) Loop indices are kept in registers and are updated at the end of each iteration. We don't store the final index value into main memory because usually the loop index is only for controlling the loop iteration. Variables for storing intermediate results are treated similarly.

(5) The index-addressing mode is used to access array elements efficiently. Loop 8 is an example where we access multi-dimensional array elements gracefully.

All these rules are very simple and can be applied to compiler-generated code. Therefore, our code also serves as a guideline for compiler writers and provides a means for comparison. The code for the single-processor mode of execution can be split naturally into two streams for the decoupled mode of execution. The access processor is responsible for memory accessing and loop control. The execute processor performs the arithmetic operations and follows the access processor's branch decisions.

Loop 3 is the simplest of all the loops we studied. The Pascal source program, the annotated single-processor code, and the decouple access/execute code are shown in Figure 4.1. The 'L' and 'S' prefixes in the opcode field designate long and short instructions, respectively. Register 7 represents either the head of the Load Data Queue (LDQ) or the tail of the Store Data Queue (SDQ). The ZBR pseudo opcode marks the location where a branch takes place if the condition tested by the antecedent Prepare to Branch (PBR) instruction is true. No code is generated for a ZBR.

The simulation results for these loops will be presented in the next section. Special coding techniques for improving system performance will also be discussed.

5. PIPE Performance for the Lawrence Livermore Loops

Using the PIPE simulator, we simulated the execution of the first twelve Lawrence Livermore Laboratories (LLL) loops [McMa72] (also see Appendix A) under various execution modes and system parameters. Section 5.1 presents the general simulation results. Program characteristics of the loops and their influence on system performance are discussed in Section 5.2. Dependence of the performance on hardware configuration and speed is studied in Section 5.3.

5.1. Parameters and Execution Modes

Two basic parameter sets and three execution modes are used in our simulation of the first twelve Lawrence Livermore loops. The parameter sets are specified in Section 5.1.1. The execution modes are discussed and compared in Section 5.1.2. Simulation results of the LLL loops in each mode and using each parameter set are also presented and briefly discussed.

```
q := Ø;
for k:= 1 to 1ØØØ do
        q := q + z[k] * y[k].
```

(a) Pascal source program

```
 1.      LENTER  1  -1ØØØ+1  /* R1 for loop index k-1ØØØ.
 2.      SXOR    2  Ø  Ø     /* R2 for q, initialized to Ø.
 3.      LLDBR   Ø  ALOOP    /* Load branch register.
 4. ALOOP
 5.      LLDLN   1  Z+1ØØØ-1 /* LDQ <-- z[k].
 6.      SIPBRLT 1  Ø  Ø     /* Loop back if R1 < Ø.
 7.      LLDLN   1  Y+1ØØØ-1 /* LDQ <-- y[k].
 8.      LADDIM  1  1        /* Increment loop index by 1.
 9.      SMULI   3  7  7     /* R3 <-- z[k]*y[k].
1Ø.      SADDI   2  2  3     /* R2 <-- R2 + z[k]*y[k].
11.      ZBR                 /* Branch taken here.
12.      LSTLN   Ø  Q        /* Generate store address for q.
13.      SMOVNFF 7  2  Ø     /* Store the value of q.
```

(b) PIPE assembly code for single-processor mode

```
access processor                    execute processor
----------------                    -----------------
1.      LENTER 1 -1ØØØ+1            1.      SXOR    2 Ø Ø
2.      LLDBR  Ø ALOOP              2.      LLDBR   Ø ELOOP
3. ALOOP                            3. ELOOP
4.      SPBRLT 1 Ø Ø                4.      SIPBRQ  Ø Ø Ø
5.      LALDLN 1 Z+1ØØØ-1
6.      LALDLN 1 Y+1ØØØ-1           5.      SMULI   3 7 7
7.      LADDIM 1 1                  6.      SADDI   2 2 3
8.      ZBR                         7.      ZBR
9.      LASTLN Ø Q                  8.      SMOVNFF 7 2 Ø
```

(c) PIPE assembly code for decoupled access/execute mode

Figure 4.1. Pascal and PIPE assembly code for LLL loop 3.

5.1.1. The Parameter Sets

Although the simulator has the capability of specifying distinct parameters for individual processors, the access processor and the execute processor in our simulation are assumed to be identical unless otherwise specified. The instruction cache has a fixed line size, 4 instruction parcels. We assume that the instruction cache is big enough to accommodate the code for each loop. In fact, 16

cache lines are enough for each of the loops except loop 8 in single-processor mode, which needs 32 lines. Two basic parameter sets are used in the simulation and are summarized in Table 5.1. The first set consists of the default parameters of the simulator. Following are the important parameters in this default parameter set.

(1) LDQ size = MLDQ size = OUTQ size = REQQ size = BRQ size = 4. Refill points for LDQ and REQQ are the same as the queue sizes; that is, a refill signal is sent as soon as an element is removed from a full queue.

(2) The memory module access time is one clock period. For such a fast memory module, there is no memory module conflict at all. Therefore, there is no need for interleaving. Including the overhead incurred in the memory control unit, the minimum effective memory delay seen by a processor is 3 clock periods.

(3) Addresses for instruction cache misses have priority in the OUTQ but not in the REQQ.

Since the basic clock period for a pipelined machine is very short, it seems unrealistic to assume that the memory module access time is only one clock period. Our second parameter set

	default	empirical
LDQ, OUTQ, BRQ	4	4
MLDQ, REQQ	4	8
LDQ-low, REQQ-low	4	4
Number of memory modules	(1)	4
Memory access time	1	4
Memory turnaround time	3	6
Instruction fetch requests have priority in OUTQ	yes	yes
Instruction fetch requests have priority in REQQ	no	yes

Table 5.1. The default and empirical parameter sets.

assumes slower memory, more modules, and longer queues. We call it the empirical parameter set. The important parameters are the following.

(1) LDQ size = OUTQ size = BRQ size = 4. MLDQ size = REQQ size = 8. Refill points for both the LDQ and the REQQ are 4; that is, the refill signal must be sent when the queue length changes from 4 to 3, or earlier.

(2) The memory module access time is 4 clock periods. The effective memory latency seen by a processor is 6 or more clock cycles. There are four interleaved memory modules. The least significant two bits of an address determine in which module the memory word resides.

(3) Addresses for instruction cache misses are given priorities in both the OUTQ and the REQQ.

A memory module conflict occurs when a request tries to access a module which is busy servicing a previous request. We increase the queue sizes to buffer more requests and to allow the issue of more load requests while the previous ones are still in process. The maximum service rate for this memory system is one request per clock period, which is achieved only if the memory modules are used in a strictly cyclic pattern.

5.1.2. The Execution Modes

Table 5.2 shows PIPE performance for the first twelve LLL loops with default simulation parameters. The execution times and the numbers of instructions executed are listed for both the single-processor (SP) and decoupled access/execute (AE) modes. The issue rate is the quotient of the instruction number to the execution time. For AE mode, the issue rates listed in the table are for the processor which executes more instructions. The last column is the ratio of execution times, and shows the speedup of AE mode over SP mode. The last row shows the total number of instructions, the total execution times, and the averages of the issue rates and speedups. In calculating the averages, the issue rates and speedups are weighted by the execution times.

In AE mode of execution, the access processor and the execute processor execute two parallel instruction streams and are capable of issuing a total of two instructions per clock period. However,

LLL loop number	SP			AE					Speedup
	#inst	exec time	issue rate	AP #inst	EP #inst	total #inst	exec time	issue rate	
* 1	4009	4069	98.5%	2406	2005	4411	2859	84.2%	1.423
* 2	8005	11053	72.4%	5604	3603	9207	6851	81.8%	1.613
* 3	6006	10039	59.8%	4005	3003	7008	5044	79.4%	1.990
† 4	5610	5697	98.5%	4592	1530	6122	5681	80.8%	1.003
† 5	7017	7425	94.5%	4011	3342	7353	5407	74.2%	1.373
† 6	7662	8065	95.0%	4665	3332	7997	5395	86.5%	1.495
* 7	3608	3687	97.9%	1686	2044	3730	2114	96.7%	1.744
‡ 8	3330	3564	93.4%	1412	1726	3138	1894	91.1%	1.882
‡*9	3909	4043	96.7%	1910	1807	3717	2002	95.4%	2.019
*10	4103	4220	97.2%	2203	2002	4205	2598	84.8%	1.624
†11	5997	8028	74.7%	4998	2000	6998	7031	71.1%	1.142
*12	5997	8028	74.7%	4998	2000	6998	6035	82.8%	1.330
	65253	77918	83.7%	42490	28394	70884	52911	81.6%	1.473

*: vectorizable by CRAY FORTRAN Compiler.
†: involves Read-After-Write hazards.
‡: uses many loop-invariant constants.

Table 5.2. PIPE performance for LLL loops, with default simulation parameters.

this important advantage of the AE mode over the SP mode is often lessened by un-even splitting of the code between the two processors. Among the twelve loops we studied, loop 4 suffers the most from unbalanced code: the access processor executes about three times the instructions that the execute processor does. Incidentally, this loop has the smallest speedup, 1.003. Loop 9 has the best balanced code between the processors, and the largest speedup, 2.019, among all the loops.

PIPE instructions use 3-bit fields to designate the queue head/tail or one of the seven foreground registers. Loops 8 and 9 use more loop-invariant constants than a register set can accommodate. Therefore, in the SP mode of execution, some constants must be read from main memory into the LDQ every time they are used. In AE mode of execution, we have one register set in each

processor and can keep more constants handy in registers. This is the reason why the total numbers of instructions for AE mode are less than those for SP mode in these two loops.

Loop 3 executing in SP mode has the lowest issue rate in the table. The main reason is that some instructions have to wait for operands. A decoupled architecture performs some code scheduling dynamically at run time and allows both processors to run smoothly. Loop 3 executes much faster in AE mode and shows a speedup of 1.99.

The loops involving Read-After-Write (RAW) hazards are marked with "†" in the table. The loops marked with "*" are classified as vectorizable by the CRAY FORTRAN Compiler. Notice that the loops involving RAW hazards are all non-vectorizable. However, the RAW hazards are not inherent in these tasks, and can easily be avoided. The detrimental effect of RAW hazards and how to avoid these hazards will be discussed later.

In order to see how memory speed influences system performance, we used the empirical parameter set and repeated the simulation in both SP and AE modes. The results are shown in Table 5.3. The table also shows the simulation results with the default parameter set for comparison. For the present, ignore the "MP exec time" lines. These will be discussed below. Generally speaking, a slower memory system degrades the performance. Loop 7 is an exception which executes quickly in either mode using either parameter set. The amount of degradation varies over a wide range, depending on the execution mode and the program characteristics. We can divide the loops into three categories: (1) hazard-bound, (2) scheduling-bound, and (3) memory-bound.

(1) Hazard-bound loops. Loops 4, 5, 6 and 11 involve RAW hazards and fall into this category. A slow memory system degrades the performance significantly for both SP and AE mode of execution. Details of the degradation mechanism will be discussed in a later section.

(2) Scheduling-bound loops. This category includes loops 1, 2, 3 and 12. Because of its dynamic-scheduling capability, the AE mode of execution suffers very little from the slow memory. On the other hand, the execution time of the SP mode increases significantly when the memory slows down. Therefore, using the empirical parameter set, loops in this category

parameter	mode	loop					
		1	2	3	†4	†5	†6
default	SP exec time	4069	11053	10039	5697	7425	8065
	MP exec time	4083	11061	10045	7229	7445	8415
parameter	AE exec time	2859	6851	5044	5681	5407	5395
set	AE-over-SP speedup	1.423	1.613	1.990	1.003	1.373	1.495
	AE-over-MP speedup	1.428	1.615	1.991	1.272	1.377	1.560
empirical	SP exec time	6483	17065	15047	8007	10453	11415
	MP exec time	8086	17091	15067	*13577	10804	11764
parameter	AE exec time	2879	6872	5066	6828	8424	8406
set	AE-over-SP speedup	2.252	2.483	2.970	1.173	1.241	1.358
	AE-over-MP speedup	2.809	2.487	2.974	1.988	1.283	1.399
parameter	mode	loop					
		7	8	9	10	†11	12
default	SP exec time	3687	3564	4043	4220	8028	8028
	MP exec time	3708	3734	4069	4247	9033	9032
parameter	AE exec time	2114	1894	2002	2598	7031	6035
set	AE-over-SP speedup	1.744	1.882	2.019	1.624	1.142	1.330
	AE-over-MP speedup	1.754	1.971	2.032	1.635	1.285	1.497
empirical	SP exec time	3722	5057	6339	6462	13030	12032
	MP exec time	4103	6798	8513	6960	*17026	12039
parameter	AE exec time	2137	3009	4212	4839	13029	6049
set	AE-over-SP speedup	1.742	1.681	1.505	1.335	1.000	1.989
	AE-over-MP speedup	1.920	2.259	2.021	1.438	1.307	1.990

†: involves Read-After-Write hazards.
*: shows interference between processors.

Table 5.3. PIPE performance for LLL loops
with various parameters and modes.

have higher speedups than other loops.

(3) Memory-bound loops. Loops 8 through 10 are in this category. The slow memory becomes the bottleneck in the system. The PIPE performance is degraded significantly for both SP and

AE mode of execution. Since the AE mode has higher memory request rate, it suffers from the slow memory more than the SP mode does: the AE-over-SP speedup decreases when the memory gets slower.

Table 5.4 shows the numbers of memory requests and the memory service rates for AE mode of simulation. The service rate is identical to the request rate, and is the quotient of the total number of requests to the execution time. The last column of the table lists the ratio of the execution times and shows the performance degradation caused by slow memory. The slowdown is negligible for the scheduling-bound loops. Using the default parameter set, the memory-bound loops have high memory request rates. However, the slow memory in the empirical parameter set can only sustain a much lower service rate. This explains the significant slowdown of the memory-bound loops. For example, the execution time of loop 9 is more than doubled with the slow memory.

In the SP mode of execution, we assumed that the memory system serviced only one processor. Another possible execution mode of a PIPE system is the Multi-Processor (MP) mode where two processors execute two independent programs and are serviced by a single memory system. The two processors then contend for memory service. Table 5.3 also shows the results of the MP mode of simulation. For each loop, we ran the same program on both processors. The two processors refer to the identical memory space, but should be considered as running two unrelated programs. Since the two processors are forced out of synchronization by the instruction cache misses, artificially introduced conflicts between them are very unlikely. However, RAW hazards may cause certain interference between the two processors' memory requests. We found two such cases, and marked them with asterisks in Table 5.3. Due to the memory contention, the execution time of MP mode is longer than that of SP mode. For the memory-bound loops, the degradation caused by slow memory is comparable for the AE and MP modes. A PIPE system can run either one program in AE mode or two programs simultaneously in MP mode. In some sense, the AE mode of execution is profitable only if the AE-over-MP speedup is 2 or more. In many applications, however, it is often preferred to be able to run one job quickly rather than to run two jobs slowly.

LLL loop number	memory requests			parameter set				slow down
				default		empirical		
	read	write	total	exec time	service rate	exec time	service rate	
1	1242	401	1643	2859	57.5%	2879	57.1%	1.007
2	2036	200	2236	6851	32.6%	6872	32.5%	1.003
3	2028	1	2029	5044	40.2%	5066	40.1%	1.004
4	1577	510	2087	5681	36.7%	6828	30.6%	1.202
5	2382	1002	3384	5407	62.6%	8424	40.2%	1.558
6	2379	999	3378	5395	62.6%	8406	40.2%	1.558
7	1138	120	1258	2114	59.5%	2137	58.9%	1.011
8	1058	240	1298	1894	68.5%	3009	43.1%	1.589
9	1377	100	1477	2002	73.8%	4212	35.1%	2.104
10	1072	1000	2072	2598	79.8%	4839	42.8%	1.863
11	2022	999	3021	7031	43.0%	13029	23.2%	1.853
12	2022	999	3021	6035	50.1%	6049	49.9%	1.002

Table 5.4. Memory service rates for LLL loops, AE mode of execution.

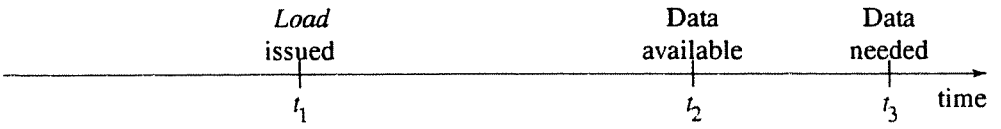
5.2. Program Characteristics

Program characteristics strongly affect performance. Load distance and branch count are discussed in Sections 5.2.1 and 5.2.2, respectively. They are the most important factors to consider in a PIPE program. Their relationship to loop size is discussed in Section 5.2.3. We also consider some optimizing techniques for small loops. In Section 5.2.4 we study the mechanism how read-after-write hazards degrade performance. Long instructions and short instructions are compared in Section 5.2.5 with respect to their implication in performance. Section 5.2.6 discuss how to access array elements effectively. Code balancing between the processors is discussed in Section 5.2.7.

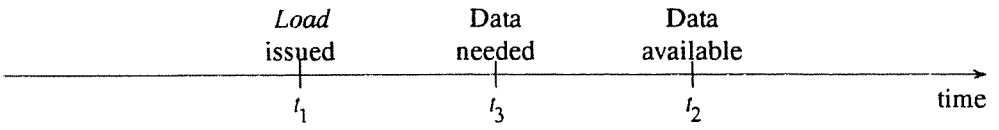
5.2.1. The Load Distance

All the operands for PIPE instructions are either registers or the head of the Load Data Queue (LDQ). The LDQ serves as a first-in-first-out buffer for data which are produced by *Load* instructions and consumed as operands by other instructions. In order to allow the consumer to proceed without waiting, we have to schedule the corresponding producer ahead of the consumer. To show the importance of proper scheduling in the SP mode of execution, we recoded loop 7 so that all the operands are loaded on demand (that is, just before they are need). With the default parameter set, the execution time increases from 3687 to 9049 clock cycles. This corresponds to a decrease in the issue rate from 97.9% to 38.9%.

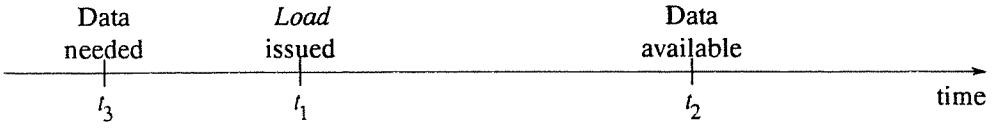
Figures 5.1 shows the timing diagrams for producing and consuming LDQ data. Assume that a *Load* instruction is issued at time t_1, the returned data enters the LDQ at t_2, and the data is needed by an instruction at t_3. The interval t_2-t_1 is the turnaround time for this load instruction. The inter-



(a) Long load distance. The consumer need not wait.



(b) Short load distance. The consumer needs to wait.



(c) Negative load distance. (AE mode only)

Figure 5.1. Timing diagrams for producing and consuming LDQ data.

val t_3-t_1 is the time allowance for the producer to complete the load operation without slowing down the consumer. Figure 5.1(a) shows the case where the operand is available when it is needed. Figures 5.1(b) and (c) show the cases where the consumer has to wait for data.

For each load instruction in single-processor code, we try to increase the number of instructions between it and the instruction that consumes the data. This number is called the *load distance* for the load instruction. Long load distances tend to increase t_3-t_1 and reduce the probability or period for the consumer to wait. However, it is not always possible to have long load distances. Branch instructions may break the code into small segments so that very few instructions can be counted in the load distance. An example is the single-processor code for LLL loop 3 shown in Figure 4.1(b), where line 9 uses the data loaded by line 7.

Code scheduling in SP mode is complicated by its dependency on the hardware configuration and the memory service discipline. Following are the important factors to consider.

(1) A sequence of load instructions with very long load distances may fill up the queues and cause deadlock. A safe instruction stream may become unsafe when the hardware configuration or characteristics change.

(2) The turnaround time for a load operation depends on the speed and utilization of the memory system. Also it may increase significantly when there are bank conflicts or instruction cache misses.

(3) The memory service discipline of the memory system also adds uncertainty to the turnaround times.

It is difficult to determine an optimal load distance for the SP mode of execution. However, if the hardware configuration is fixed, then we can schedule load instructions ahead as far as the queue sizes allow. By doing this, we have achieved very high issue rates in SP mode for many of the LLL loops.

In decoupled access/execute mode of execution, the code scheduling is between the instruction streams for two processors instead of within a single instruction stream. Scheduling takes place automatically at run time and suffers very little from the three problems mentioned above. When the producer (that is, the access processor) of data runs too far ahead of the consumer (that is, the execute processor), it is perfectly legal to block the access processor and allow the execute processor to proceed. Since the bottleneck is at the execute processor under this condition, blocking the access processor does not degrade system performance. The intent of a decoupled architecture is that the access processor should run ahead of the execute processor and reduce or eliminate observed memory delay. Any temporary variation in the memory turnaround time should be smoothed away in the queues.

In fact, in most of the loops we studied, the access-processor code has more instructions than the execute-processor code; therefore, the access processor does not run ahead of the execute processor. The timing diagram for the load instructions in these loops looks like the one in Figure 5.1(c). The figure, which is unique to the AE mode of execution, depicts the situation where the data are needed by the execute processor while the corresponding load instruction in the access processor has not yet issued. Therefore, the LDQ is empty most of the time; a data word is consumed by the execute processor as soon as it enters the LDQ. Our simulation results show that the average LDQ length is very small for the execute processor. The system throughput for these loops is limited by the access processor. We believe that the work load for the execute processor will increase when we add more stages to the ALU pipeline and add floating-point operation to the PIPE instruction set.

5.2.2. The Branch Count

The Prepare to Branch (PBR) instructions in the PIPE architecture are designed to allow smooth control flow in program execution. It is important for a PBR instruction to have a large branch count so that enough instructions following it can proceed without knowing the branch decision. A simple technique is used in coding the LLL loops. We update the loop index at the end of the loop (that is, after the PBR instruction) to increase the branch count. Care is taken in

formulating the termination condition and in using the loop index as an index register. An example is the access-processor code for loop 3 shown in Figure 4.1(c), where line 4 is the PBR instruction and line 7 updates the loop index. The branch count would be 4 parcels instead of 6 if the loop index is updated at the head of the loop.

In order to fully utilize the processor, we should try to maintain a continuous flow of instructions through the pipeline stages. Holes in the instruction flow reduce the system throughput. Referring to the structure of the processor shown in Figure 3.1, we can count the number of instructions a branch count should cover in order to avoid holes in the instruction flow. Suppose that the first stage of the ALU executes a PBR instruction and determines that the branch condition is true. To avoid holes, there should be one instruction at

(1) the Instruction Queue (IQ),
(2) the decoder,
(3) the issue logic, and
(4) the first stage of the ALU.

while the instructions at the branch target are being fetched into the Cache Register (CR). Therefore, if a branch is taken, the branch count should cover four or more instructions in order to avoid holes in the instruction pipeline and the ALU pipeline. If the branch is not taken then the branch count should cover three or more instructions.

Some PBR instructions in the programs are not able to have a large branch count unless special coding techniques are used. The access-processor code in Figure 4.1(c) shows an example of a very small loop body that contains only three instructions (excluding the PBR instruction). The branch is taken in all iterations except the last one. The small branch count accounts for the low issue rate of loop 3 in the AE mode of execution.

The branch count is specified by a 3-bit field in a PBR instruction. A problem with this format is that the maximum branch count, 7, cannot cover 4 instructions if they are all two-parcel instructions. In our experience, most of the access-processor instructions are long *load* instructions. The branch count is 6 and covers three long instructions in the example above. If we add a long instruc-

tion to the loop, it must be placed before the PBR instruction; if placed after, the branch count would be 8, greater than the maximum. In order to improve the issue rate, it is desirable to be able to specify a branch count of 8. One easy way to do this is to interpret the 3-bit branch count 0 as 8. If there is nothing to go after the PBR instruction then we pad a no-op instruction after the PBR and make the branch count 1. The no-op instruction only increases the code size by one byte and is executed free because the ALU is idle otherwise. Of course, for a different implementation some other maximum branch count may be desirable

5.2.3. The Loop Sizes

The hit ratio of the instruction cache is very high in all of our simulation because the instruction cache is big enough to hold all instructions of the loop body. For a direct-mapped instruction cache, the hit ratio begins to decrease linearly when the size of the inner-most loop exceeds the cache size. It vanishes when the loop size is twice the cache size. However, large inner loops are very rare in real applications.

On the other hand, small loops are used very often and can degrade the performance significantly if not properly coded. Small loops have the following disadvantages:

(1) The branch count of the PBR instruction is small.
(2) The load distances are short.
(3) The overhead for flow control is high.

The first two disadvantages are unique to the PIPE architecture and have been discussed earlier. The overhead for flow control includes updating the loop index and executing the PBR instruction. In the single-processor code for loop 3 shown in Figure 4.1(b), the loop body contains only six instructions (lines 5 through 10). The overhead (lines 6 and 8) is one third of the total instructions executed in each iteration.

Loops 3, 11 and 12 are very small. Following are some other typical uses of small loops:

(1) initializing an array,
(2) constructing a free list,
(3) computing the sum of array elements,
(4) searching for a specific value in an array.

Since small loops are used very frequently, it is important to execute them effectively. Here we consider two coding techniques for improving the issue rate of small loops: (1) software pipelining, and (2) code doubling.

The execution of loop 3 in the single-processor mode can be represented as

$$(L_{11}L_{12}WU_{11}U_{12}O)(L_{21}L_{22}WU_{21}U_{22}O)\cdots(L_{n1}L_{n2}WU_{n1}U_{n2}O)$$

where L_{ij} is the j-th load instruction in the i-th iteration, U_{ij} uses the data loaded by L_{ij}, W and O stand for 'Waiting' and 'Overhead', respectively. The waiting states are necessary because the load distances are very short. The parentheses delineate the loop structure. The loop is iterated n times.

The software pipelining technique [CoSt81, BrWe83] ravels the loop, re-arranges and regroups the instruction stream into

$$L_{11}L_{12}(L_{21}L_{22}U_{11}U_{12}O)(L_{31}L_{32}U_{21}U_{22}O)\cdots(L_{n1}L_{n2}U_{n-1,1}U_{n-1,2}O)U_{n1}U_{n2}.$$

The loop body remains the same size. However, including the prologue and the epilogue, the static code size is about doubled. The new loop body loads data for the next iteration and uses the data loaded by the previous iteration. This new structure effectively increases the load distances and, hopefully, eliminates the waiting states. The attainable branch count and the overhead for flow control remain about the same.

Another technique for improving the issue rate of small loops is to aggregate k consecutive iterations into a larger loop body. For $k=2$, this ''code doubling'' technique yields the following structure:

$$\textit{prologue}\ (L_{11}L_{12}L_{21}L_{22}U_{11}U_{12}U_{21}U_{22}O)\cdots(\cdots).$$

The prologue is used to handle special cases where the number of iterations is zero or odd. If n is odd then the first iteration of the original loop is executed in the prologue. The static code size is more than doubled. If the number of iterations is known to be even then the prologue can be omitted. The runtime overhead of the prologue is negligible when n is big. The new structure improves the issue rate by overcoming all the disadvantages listed above: it reduces the flow control overhead and increases the branch count and the load distances. We applied this technique to loop 3 and

simulated the SP mode of execution with the default parameter set. A comparison with the original results shows great improvements. The total number of executed instructions decreased from 6006 to 5016; the execution time decreased from 10039 to 5563; and the issue rate increased from 59.8% to 90.2%.

When splitting the single-processor code into two cooperating streams, we end up with two smaller loops. The overhead for flow control becomes more significant in the access processor. The execute processor has the overhead of executing the "Prepare to Branch from Queue" (PBRQ) instructions. The branch count for either the PBR or the PBRQ instructions is smaller than that in the SP mode. For loops 2, 3 and 12, the low issue rates in AE mode are caused by small branch counts in access-processor code. The code doubling technique can be applied to the decoupled mode as well.

5.2.4. The Read-After-Write Hazards

Many of the loops involve Read-After-Write (RAW) hazards and are marked with "†" in Tables 5.2 and 5.3. Generally, PIPE's performance of these loops are degraded by the hazards. Loops 11 and 12 are syntactically identical (see Appendix A) and provide us a good opportunity to see the effect of RAW hazards on performance. We now use loop 11 as an example and study the bad effect of RAW hazards in detail. The Pascal source code looks like

```
for k := 2 to 1000 do
    x[k] := x[k-1] + y[k].
```

Figure 5.2 illustrates part of the activities during AE mode of execution of loop 11. The arcs in the figure denote the precedence relations among activities. For example, node 2 can execute only if both nodes 1 and 1′ have finished. Nodes on a directed path must be executed in order. A *critical path* between two nodes is a directed path that has the longest execution time. For loop 11, the following nodes are on a critical path.

(1) At the end of an iteration, the execute processor sends the new value of x[k-1] to the memory control unit.

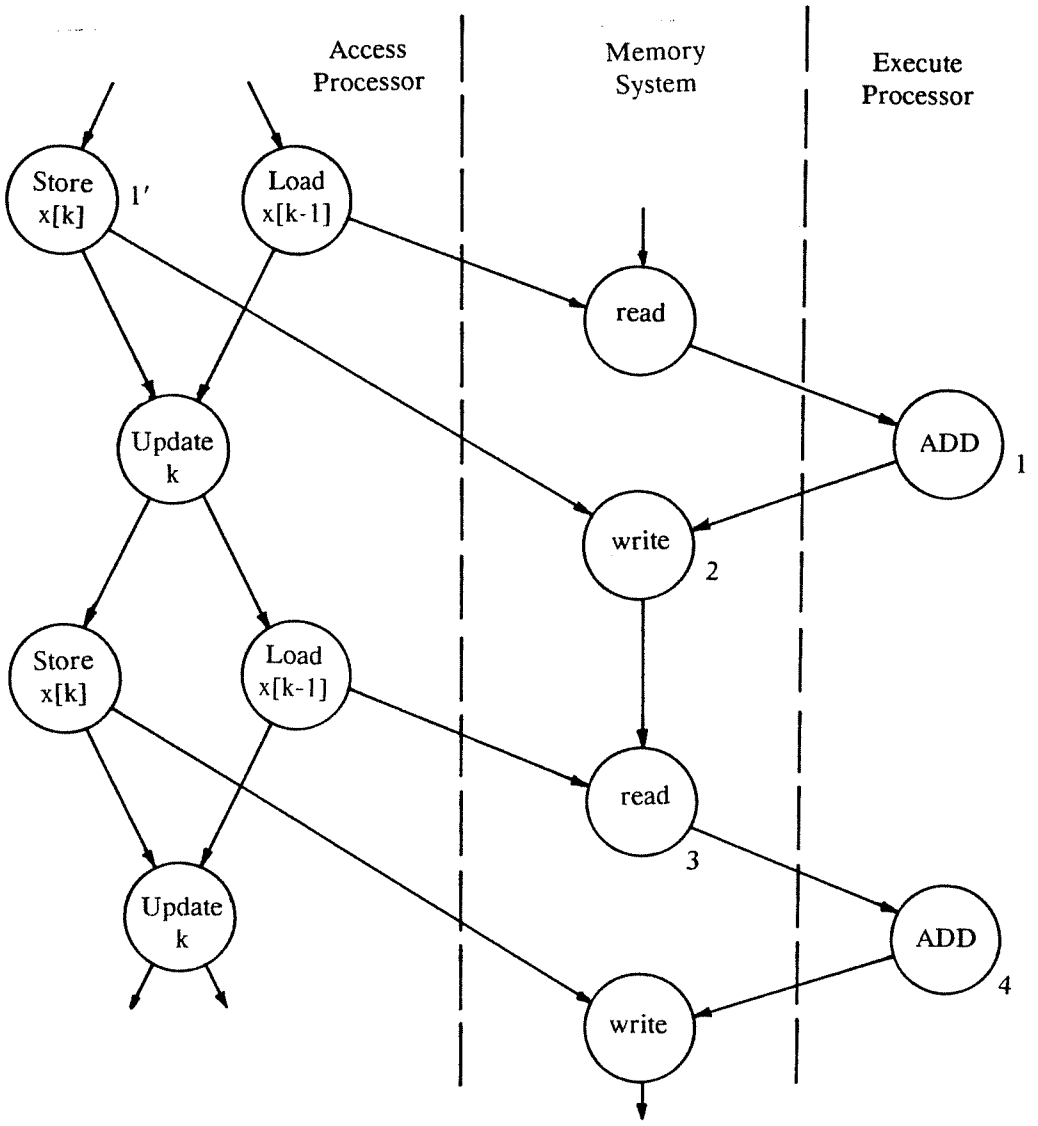


Figure 5.2. Precedence relations among activities in LLL loop 11.

(2) This value is written to a memory location.

(3) Due to the RAW hazards, the read operation can proceed only if the write operation has finished. Upon completion of the read operation, the value of x[k-1] is sent to the execute processor.

(4) Once the operands enter the LDQ, the execute processor can issue the ADD instruction that computes x[k-1]+y[k].

Nodes 2 and 3 access the same memory word and must be executed in the read-after-write order. For loop 12, nodes 2 and 3 can proceed in any order because the read address is different from the write address. This is the essential difference between loops 11 and 12.

With the default parameter set, this critical path takes 7 clock periods to execute. Loop 11 traverses 999 similar paths and has an execution time close to 7000 clock periods (see Table 5.3). The execution times of both nodes 2 and 3 include the access time of the memory modules. When we change from the default parameter set to the empirical parameter set, the execution time of each node increases by 3 clock periods. Therefore, the execution time of the critical path increases form 7 to 13 clock periods; the total execution time is about 13000 clock periods. Incidentally, the SP mode of execution with the empirical parameter set has the same critical path and the same execution time (see Table 5.3).

One way to reduce the total execution time of loop 11 is to break the critical path. At the end of an iteration, the execute processor can save the newly calculated value of x[k] in a register as well as send it to the memory control unit. The register then can supply an operand for the ADD instruction in the next iteration. Therefore, node 3 is eliminated and the critical path is broken. Row 3 of Table 5.5 shows the simulation result when this technique is applied. This loop can be further optimized by other techniques. Row 4 shows the effect of doubling the loop size, as was discussed in

	parameter set	RAW avoided	code doubled	short ADD	AP #inst	EP #inst	total #inst	exec time	issue rate	speedup
1	empirical				4998	2000	6998	13029	38.4%	1.00
2	default				4998	2000	6998	7031	71.1%	1.85
3	empirical	X			4000	3000	7000	6057	57.1%	2.15
4	empirical	X	X		3000	2499	5499	3571	84.0%	3.65
5	empirical	X	X	X	3001	2499	5500	3092	97.1%	4.21

Table 5.5. PIPE performance for LLL loop 11 in AE mode of execution, with various optimizing techniques applied.

Section 5.2.3. Row 5 shows the effect of using short ADD instruction to update the loop index. It will be discussed in Section 5.2.5. The last column of Table 5.5 lists the speedup in execution time with respect to the first row. Row 2 shows a speedup of 1.85 when a fast memory system as specified by the default parameter set is used. In the last row, we applied three optimizing techniques and achieved a speedup of 4.21.

Sometimes it is not easy to break the critical path. For example, consider the Pascal code of loop 4 (see Appendix A). In line 6, the x[lw] on the right hand side might be the same as the left hand side that got updated in the previous iteration. This alias problem also forces us to actually store the value of x[L-1] into main memory in each iteration unless some special analysis is done. Another way to reduce the effect of RAW hazards is to reduce the execution time of the critical path by short-circuiting the read-after-write operation. A sophisticated memory control unit can recognize the read-after-write operation. Upon receiving a value from node 1 (see Figure 5.2), the memory control unit can send the value to node 4 without first waiting for the write operation to finish and then actually carrying out the read operation. This capability is not implemented in our simulator.

5.2.5. Long Instructions versus Short Instructions

The PIPE architecture has two instruction formats: long 32-bit instructions and short 16-bit instructions [SPKG83]. In a short instruction, there are three 3-bit operand fields. The three operands are typically two sources and one destination. In a long instruction, there is a single 3-bit operand field, and a 22-bit signed displacement field to hold immediate data or address offset. Unlike the CRAY-1 [Cray79] computers, a PIPE processor can issue an instruction, short or long, in one clock period. Although many operations on data can be accomplished in either format, the choice of long or short instructions can affect system performance.

Use of long instructions tends to increase program size. Program size is of least importance when a program is stored in the secondary storage. It becomes a concern when the program is loaded into main memory for execution. The instruction cache makes the program size a very important factor to consider. Compact code is preferred since it takes less cache misses to get loaded

into the instruction cache. Also a smaller loop has a better chance to completely fit in an instruction cache and enjoy a high hit ratio.

The smallest instruction size in the PIPE architecture is a 16-bit parcel. Since a long instruction is not required to begin at an even-parcel address, it can straddle two cache lines. Fetching such a line-crosser needs two reads from the instruction cache. In Section 5.2.2 we pointed out that if a branch is taken then the branch count of the PBR instruction has to cover at least four instructions to avoid holes in the instruction and ALU pipelines. If the branch target is a line-crosser, then the branch count has to cover at least five instructions. The maximum branch count of 7 parcels can at most cover 3 long instructions. This is another reason short instructions are preferred.

In our original hand coding of the loops, long instructions with immediate operands are use to update the loop index. Alternatively, we can keep the increment in a register and use a short instruction to accomplish the same work. Rows 4 and 5 of Table 5.5 show an example in which the PIPE performance is improved by replacing a long ADD instruction with a short one. For row 4, there are four long instructions, including the long ADD instruction that updates the loop index, at the end of the loop. The branch count covers only three of them. The low issue rate is caused by the insufficient coverage of the branch count. For row 5, the long ADD instruction is replaced by a short one. The branch count, 7, covers this short instruction and three other long instructions.

Since registers are a scarce resource, it may not be practical to dedicate a register to a special purpose such as holding the index increment. It has been shown in many studies [AlWo75, Tane78] that most of the constant operands used in programs are very small. This is indeed the case for our index increments. Therefore, it is desirable to have a short instruction format which allows us to specify a small immediate operand. A plausible format is to combine two of the 3-bit fields into a small immediate operand field.

The CRAY-1 computers use another method to specify some frequently used constants [Cray79]. When some registers are referenced in certain fields of certain instructions, the contents of the respective register are not used; instead, a special operand is generated. Both this and the

above methods complicate the decoder logic and we may need two stages of the instruction pipeline to completely decode an instruction. A side effect of a longer pipeline is that the branch count has to cover more instructions. Further study is necessary in order to determine which method is better and whether it can noticeably improve PIPE performance.

5.2.6. Accessing Array Elements

Long instructions with immediate offsets are often used to load or store array elements. The code in Figure 4.1 contains many such instructions. In order to keep the code compact, however, short load or store instructions are preferred. Both the short and long load/store instructions have the auto-incrementing capability which is useful in accessing array elements at a fixed stride. The u array in loop 7 and the px array in loops 9 and 10 (see Appendix A) can be accessed effectively by short load/store instructions with auto-incrementing.

Accessing the elements of a multi-dimensional array is more complicated and often requires some arithmetic operations to calculate the offsets. However, we can often recognize common sub-expressions and use them without recalculation. Loop 8 operates on three 3-dimensional arrays of same structure. In each iteration, we compute the offset only once for the subscripts [kx, ky, 1]. The offset for all other subscript combinations differ from this value by constant amounts which are independent of the input data. Therefore, all operands can be loaded with very little overhead. However, it takes an extremely good compiler to recognize this common sub-expression and generate compact code comparable to our hand-written code.

5.2.7. Code Balancing

Table 5.2 shows that, in the AE mode of execution, the work loads for the two processors are different. The access processor generally executes more instructions than the execute processor. Loop 4 has the most un-even split of work load; the ratio is about 3 to 1. Since the types of the work done by the two processors are mutually exclusive, there is no easy way to redistribute the work load between the processors.

Table 5.5 shows some limited amount of redistribution of work load as side effects of the application of some optimizing techniques. The instruction ratio in the second row is about 5 to 2. By moving one instruction from the access-processor code to the execute-processor code, we obtained a better ratio of 4 to 3 in the third row. The code-doubling technique reduced the overhead of loop control. The savings was more significant in the access processor. This led to a better instruction ratio, 6 to 5, in the last two rows.

Depending on the task at hand, we can also make special arrangements to reduce the work load of the access processor. For example, the following statement is commonly used to construct a free list

for i:= 1 to n do next[i]:=i+1.

This is a typical loop where the loop index is also used as an operand for the execute processor. Conventionally the access processor has to load the operand i for the execute processor. This implies possible Read-After-Write hazards because, in each iteration, the access processor has to write the loop index i into main memory and then read it back for the execute processor. We can let the processors cooperate in a looser manner and get the job done much more easily. The goal is to supply n store addresses from the access processor and n data from the execute processor to the memory control unit. Each processor can keep its own value of i in a register and execute its loop independently of the other processor. We believe that this technique can be incorporated in a compiler.

5.3. Hardware Characteristics

We are interested in finding a cost-effective configuration of a PIPE system. Various architectural queues are discussed in Sections 5.3.1 through 5.3.4. The effect of the queue sizes on PIPE's performance is studied. Section 5.3.5 shows how to avoid result bus conflicts by dynamic scheduling. The utilization and configuration of the busses between the processors and the memory system are studied in Section 5.3.6. Section 5.3.7 shows how memory access time affects PIPE performance in both execution modes.

5.3.1. The Output Queue

The operation results and memory requests from a processor are transmitted to the memory control unit through the output bus. The transmission may be delayed due to the following reasons.

(1) The *LDQ-full* message is urgent for flow control and is given higher priority to use the bus. The *LDQ-refill* message also gets priority when the LDQ is drained down to a prespecified length.

(2) If many previous requests are still waiting in the request queue, then the memory control unit may not have space for new requests. The *REQQ-full* message prevents the processor from sending more requests.

The Output Queue (OUTQ) serves as a buffer between the ALU and the output bus. A short OUTQ is enough to buffer the requests delayed by the first reason above. The second case above usually implies that the system bottleneck is either in the memory system or in the other processor. The OUTQ will eventually get filled up no matter how long the hardware queue is. The solution to this problem is to locate and break the bottleneck instead of increasing the OUTQ size.

Both the default and the empirical parameter sets have a short OUTQ of size 4. The simulation results of the LLL loops show that, for both the SP and the AE mode of execution, the OUTQs are usually very short unless there are bottlenecks in the memory system caused by Read-After-Write (RAW) hazards or bank conflicts. In the AE mode of execution, the access processor usually has a longer average queue length because it does many alternate loads for the execute processor. For loops 7 and 8, the access processor executes less instructions than the execute processor does. These are the only loops where the access processor races ahead of the execute processor and fills up its own OUTQ and the execute processor's LDQ.

The architectural queues play a very important role in the PIPE architecture. To work properly, each queue must meet some special requirements. For the OUTQ there are two important requirements.

(1) It should be able to supply output and receive input simultaneously.

(2) If the queue is initially empty, it should be able to pass the input to the output within the same clock cycle.

The first requirement is obvious. The second one becomes important, especially in the SP mode of execution, when the load distances (see Section 5.2.2) are not long. An option of the simulator disables the pass-through capability of the OUTQ and requires each element to stay in the queue for at least one clock cycle. Since the OUTQ is empty most of the time, this option effectively increases the turnaround time of each load operation by one clock cycle. In loop 1, the load distances are long enough for the default parameter set so that setting this option does not affect the execution time. The load distances become too short when the empirical parameter set is used. Simulation results show that setting this option increases the execution time of loop 1 by 405 clock cycles, which amounts to about one extra clock cycle per iteration of the loop.

In conclusion, we find that a short OUTQ of size 4, as was specified in both parameter sets, suffices for efficient execution of the loops we studied. Very little improvement can be achieved by using a longer OUTQ. On the other hand, system performance may be degraded significantly, especially in the SP mode of execution, if the OUTQ fails to meet the hardware specification.

5.3.2. The Request Queue

The memory control unit buffers all the memory requests in queues. Figure 3.2 proposed a symmetrical configuration of the buffering stages in the memory control unit. The data/instructions read by the left IAQ/LAQ are returned to the access processor. Those from the right ones go to the execute processor. Although two inputs to each of the LAQs and the SAQs are shown in the figure, only one input to each queue is active in any of the three execution modes. Therefore, a conventional single-input single-output queue suffices; no special design of the queue is needed. For the AE mode of execution, the left six queues receive requests from the access processor. Requests from the execute processor only go to the SDQ and the IAQ on the right. When executing in either the SP or the MP mode, a processor only uses its own four queues and does not load or store on behalf

of the other processor.

To avoid system deadlock, the memory control unit needs to be able to block one processor while letting the other one proceed. For simplicity, the simulator sends *REQQ-full* or *REQQ-refill* message to a processor depending on the processor's total outstanding memory requests. Other flow control mechanisms are also possible. Using the buffer queues shown in Figure 3.2, the memory control unit can accept requests from a processor until any of its receiving queues is full; the *REQQ-full* message then has to be sent. This implies that any full queue will block the CPU-to-MCU transmission even if the next intended receiving queue is not full. To better utilize the buffer queues, the memory control unit can send the status of these queues instead of the simple boolean *REQQ-full* and *REQQ-refill* flags. The processor uses the status to determine whether the next OUTQ element can be sent.

5.3.3. The Load Data Queue

The Load Data Queue (LDQ) in a processor and the Memory Load Data Queue (MLDQ) in the memory control unit together serve as a buffer for the operands loaded by the *Load* instructions. Both the default and the empirical parameter sets have an LDQ of size 4, which seems to work fine. Instructions for cache misses and the messages for flow control both share the MCU-to-CPU bus with the LDQ data. The simulator counts the numbers of each kind of transmission, but does not measure the contention among them.

There are two different interpretations of an instruction which uses the LDQ for both of its operands. One interpretation is to take one element from the LDQ and use that value as both operands. The current PIPE interpreter has another interpretation which takes two elements from the LDQ as operands. In accordance with the interpreter, the PIPE simulator assumes the capability of taking two elements from the LDQ in one clock period. To meet this requirement, special efforts in the circuit design of the queue are necessary. If the design is too difficult then we may have to restrict ourselves to the first interpretation. Another plausible solution is to extend the PIPE architecture and build two LDQs in each processor.

Some of the hand coded loops contain instructions that use the LDQ for both operands. Each of these instructions can be replaced by two instructions so that each instruction only uses one element from the LDQ. In most of the loops, this change will not degrade system performance because the execute processor originally has a lighter work load.

In the simulation of the loops, the Memory Load Data Queue (MLDQ) in the memory control unit has a very small average length, unless the access processor races ahead of the execute processor. We specified the size of the MLDQ to be 8 in the empirical parameter set to allow a smooth flow of the loaded data. The required length is a function of the memory access time based on a worst case consideration. The memory control unit reserves a space in the MLDQ before issuing a load request. The space is released when the MLDQ element is transmitted to a processor. In the empirical parameter set, we need an MLDQ size of at least four to allow the consecutive issue of four load requests. In general, if the access time of the memory modules is n clock cycles, then the MLDQ should be a little longer than n.

5.3.4. The Branch Queue

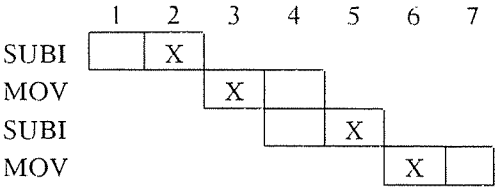
In the hand coding of the first twelve LLL loops, we use only the AP-to-EP Branch Queue (AEBRQ). The average length of the branch queue is very short for every loop. This is obvious in the loops where the access processor does not race ahead of the execute processor. In the other loops, the access processor is blocked by full OUTQ instead of by full BRQ since many operands are loaded in each iteration. Therefore, a short BRQ is adequate for all applications.

When a very short OUTQ and REQQ are used in the AE mode of simulation, we find that the PIPE system enters a deadlock state in loop 10. The AP code has many alternate store instructions in front of a Prepare to Branch (PBR) instruction while the EP code has a Prepare to Branch from Queue (PBRQ) instruction whose branch count covers many instructions which generate store data. Since many store addresses are still waiting for the matching SDQ elements, the memory control unit cannot accept more requests from the access processor. In the access processor, the PBR instruction is blocked by the preceding alternate store instructions which in turn are blocked by the full OUTQ.

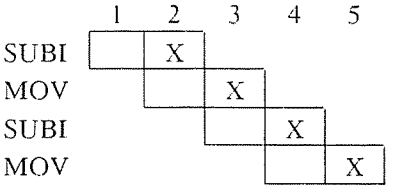
The execute processor cannot generate SDQ elements because the preceding PBRQ instruction is blocked by the empty EABRQ. This deadlock is caused by an error in the issue logic of the simulator. The PBRQ instruction should not be blocked by an empty input BRQ. Instead, the instruction beyond its branch count should wait for the branch decision from the BRQ. With this change, the execute processor can generate the awaited SDQ elements and break the deadlock situation.

5.3.5. Result Bus Scheduling

A PIPE processor has a two-stage pipelined Arithmetic Logic Unit (ALU). Both stages of the ALU can send data through the result bus to the register file or the OUTQ. A reservation table for the result bus is required to avoid conflicts between the two ALU stages. Figure 5.3(a) shows the reservation table for a segment of the execute-processor code for loop 10. The integer subtract (SUBI) instruction uses the result bus to store the result from the second stage of the ALU. The move (MOV) instruction uses the result bus at the first stage of the ALU. The first MOV instruction cannot be issued at clock cycle 2 due to bus conflict. However, if the MOV instruction postpones its use of the result bus to the second stage of ALU then it can be issued without any conflict. Figure



(a) without dynamic bus scheduling.



(b) with dynamic bus scheduling.

Figure 5.3. Reservation tables for the result bus.

5.3(b) shows the reservation table for the same instruction sequence after scheduling the uses of the result bus. The result bus scheduling saves one clock cycle for every two instructions in this example. Figure 5.3 shows that all the bus conflicts are resolved by scheduling. However this is not always true. The post-incrementing load/store instructions use the result bus in both ALU stages and cannot be scheduled.

A methodology for resolving collisions in pipelined architecture can be found in [PaDa76]. Since the ALU has only two stages, the scheduling algorithm is very simple and can be implemented in hardware. Both parameter sets assume the capability of dynamic bus scheduling. When this capability is turned off, the execution time of loop 10 changes from 2598 to 2995 clock cycles, a 15% increase, in AE mode of execution. Similar, but smaller, increases are observed in the simulation of other loops. We feel that the hardware implementation of this capability can pay off well.

5.3.6. The Bus Utilizations

Figure 3.1 shows that there are four unidirectional busses between the Memory Control Unit (MCU) and the processors: AP-to-MCU, MCU-to-AP, EP-to-MCU, and MCU-to-EP. Table 5.6 shows the utilizations of these busses in both the SP and AE mode of simulation of the LLL loops with the default parameter set. The words transferred through a bus can be data, addresses, instructions, or control messages. The utilization of a bus is the quotient of the number of total transferred words to the total execution cycles.

For each store operation, one address and one data word are transmitted to the memory control unit, but nothing is returned. This is the reason the CPU-to-MCU bus has a higher utilization than the MCU-to-CPU bus in the SP mode of simulation. Loop 3 computes the inner product of two vectors and stores the value only once after all iterations of the loop. For this loop, the MCU-to-CPU bus has a higher utilization because a cache-miss request returns a whole line of instructions.

In the AE mode of simulation, the access processor executes many alternate load and alternate store instructions. An address is transmitted to the memory control unit for each of such instruction. This explains the high utilization of the AP-to-memory control unit bus. Each alternate load

LLL loop number	SP		AE					
	to MCU	from MCU	to MCU			from MCU		
			AP	EP	total	AP	EP	total
1	49.4%	30.3%	56.3%	14.2%	70.4%	0.8%	42.6%	43.4%
2	21.8%	18.3%	32.2%	3.0%	35.2%	0.4%	29.4%	29.7%
3	20.0%	20.1%	39.8%	0.1%	39.8%	0.3%	39.9%	40.2%
4	44.9%	27.5%	36.0%	9.0%	45.0%	0.7%	27.1%	27.8%
5	58.7%	32.1%	61.9%	18.6%	80.5%	12.8%	43.5%	56.3%
6	53.8%	29.4%	61.9%	18.6%	80.5%	12.8%	43.5%	56.3%
7	49.3%	30.6%	57.2%	28.4%	85.6%	12.5%	52.3%	64.8%
8	57.3%	37.5%	62.5%	33.7%	96.2%	8.6%	51.4%	60.0%
9	57.5%	39.1%	70.7%	25.1%	95.9%	2.3%	66.6%	68.9%
10	90.5%	25.3%	77.4%	46.3%	123.8%	17.2%	39.6%	56.8%
11	49.8%	25.1%	42.7%	14.2%	56.9%	28.4%	28.5%	56.9%
12	49.8%	25.1%	49.7%	16.6%	66.3%	0.3%	33.2%	33.5%

Table 5.6. Bus utilizations for LLL loops, with default simulation parameters.

instruction sends an operand to the execute processor and accounts for most of the MCU-to-EP bus utilization. Each store address is matched with a data word from the execute processor. These data words account for most of the EP-to-MCU transmissions. In the simulation of loops 1 through 6, 11, and 12 with default parameter set, we observe that the utilizations of the AP-to-MCU bus is approximately the sum of those of the EP-to-MCU and MCU-to-EP busses. This relation is obscured by control messages in the other loops. The MCU-to-AP transmissions mostly consist of instructions and control messages.

In the VLSI implementation of the PIPE processors and the memory control unit, a limit is posed on the number of pins available on an integrated circuit package. A PIPE processor uses dedicated input and output pins to avoid the time penalty of multiplexing the pins. The memory control unit serves two processors and the limit on the number of pins becomes more stringent. One way to

reduce the number of pins is to combine some of the busses. Table 5.6 shows the total utilizations of the CPU-to-MCU busses and the MCU-to-CPU busses to help evaluating whether such combinations are feasible.

The total utilizations of the CPU-to-MCU busses are very high for most loops. The combined utilization for loop 10 is 123.8%, which means that such a combination will definitely degrade the performance. Since the bus is driven by both processors, some kind of arbitration between the processors is required. The arbitration mechanism may reduce the effective capacity of the bus by assigning the bus to an idle processor while the other processor is waiting for transmission. Also there may be some time penalty associated with switching the drivers of the bus.

Combining the MCU-to-CPU busses seems to be more plausible. No arbitration between the processors is necessary because they are receivers instead of drivers. Based on its queueing discipline, the memory control unit selects an element from a non-empty MIQ or MLDQ (see Figure 3.1) to transmit. The intended receiver of a transmission is designated by the tag associated with each transmission. The last column of the table shows that the total utilizations of the combined bus are not very high. Temporary bus contentions are buffered in the MIQ and the MLDQ. However, congestion of the combined bus is still possible if the processors transmit in bursts and if the bursts collide. Further investigation is necessary to determine whether such collisions have nontrivial probability.

5.3.7. The Memory Modules

In Table 5.4 we saw that slow memories degraded the PIPE performance of many loops. We have also seen that, with the default parameter set, the memory request rate can be as high as 80%. If an m-way interleaved memory system consists of memory modules whose access time is n clock cycles, then the system has a maximum service rate of m/n requests per clock cycle. The service rate decreases if there are memory module conflicts among the requests. As a rule of thumb, we need the number m to be equal to or greater than n in order to cope with the high request rate in the PIPE system.

The memory access time is an important parameter of a computer system. Slow memories significantly degrade the performance of conventional computers which execute instructions one after another in a strict order of instruction fetching, operands fetching and result storing. The architectural queues in a PIPE system are designed to hide the memory access delays and to allow the smooth execution of the instruction streams. Tables 5.7 and 5.8 list the execution times of the first Lawrence Livermore loop in the SP and AE mode of simulation, respectively. The execution times are plotted in Figures 5.4 and 5.5. Unless otherwise specified, the default parameters are used. The access time of the memory modules is varied over a wide range from 1 to 16 clock cycles. Various combinations of the LDQ and MLDQ lengths are used. We assume that there are infinitely many modules so that no memory module conflicts occur.

Queue sizes		Memory Access Time				
LDQ	MLDQ	1	2	4	8	16
2	2	4070	4875	6483	9699	16139
3	3	4070	4476	5288	6912	10160
≥4	≥4	4069	4475	5287	6911	10159

Table 5.7. Single-processor execution times for LLL loop 1, varying memory access time and queue sizes.

Queue sizes		Memory Access Time				
LDQ	MLDQ	1	2	4	8	16
2	2	2859	2865	3677	6097	10937
3	3	2859	2864	2878	4100	7340
4	4	2859	2864	2878	3206	5555
4	6	2859	2864	2878	2906	3758
≥4	≥8	2859	2864	2878	2906	2963

Table 5.8. Decoupled mode execution times for LLL loop 1, varying memory access time and queue sizes.

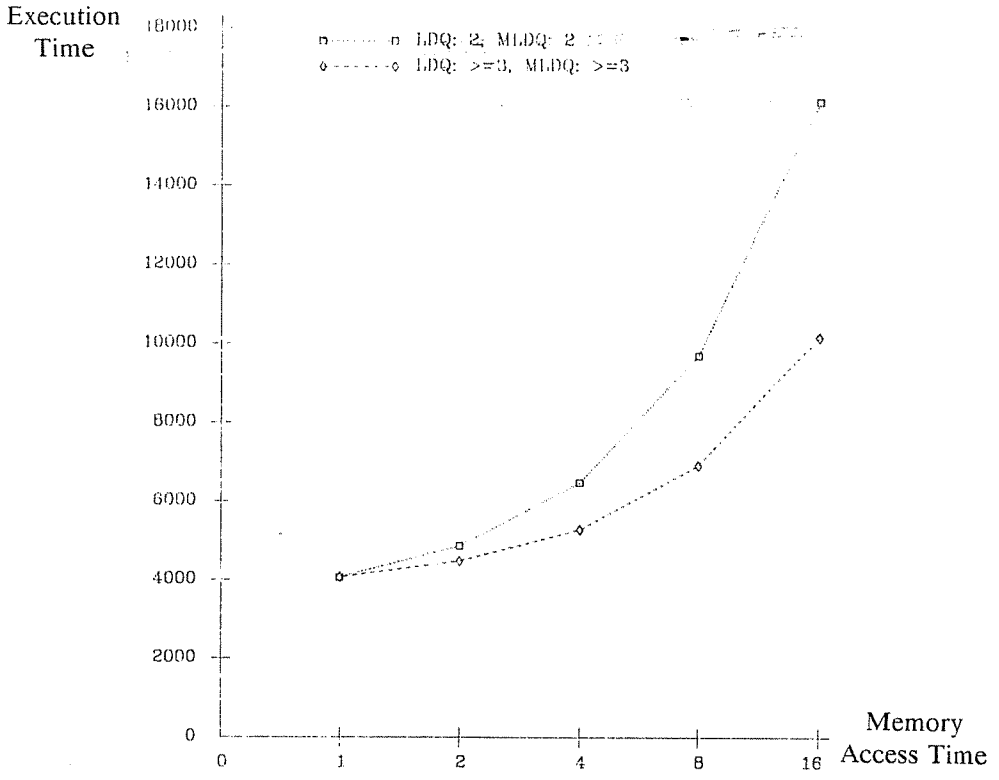


Figure 5.4. Loop 1 SP mode execution time vs. memory access time, varying queue sizes.

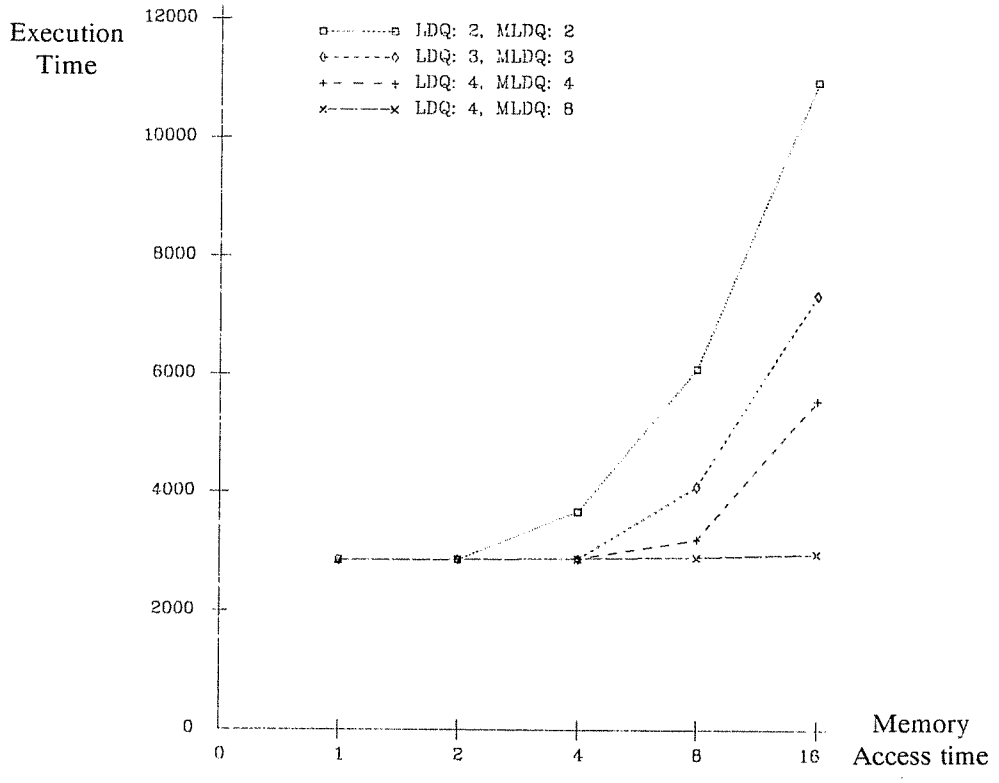


Figure 5.5. Loop 1 AE mode execution time vs. memory access time, varying queue sizes.

For the SP mode of execution, Figure 5.4 shows that, as expected, the execution time increases when the memory gets slower. In Figure 5.5, the AE mode of execution with short queues also shows the same trend. By increasing the sizes of the LDQ and the MLDQ, the execution times in both execution modes can be improved to a certain lower bound. The main difference between the two figures lies in the shape of the lower bound. In the SP mode of simulation, the lower bound increases with the memory access time. This means that the degradation in the performance cannot be avoided by using longer queues. In the AE mode of simulation, however, the lower bound is essentially flat. This means that the long memory delay can be hidden by using longer LDQ and MLDQ. The advantage of the AE mode over the SP mode becomes more significant when the memory gets slower. The ratio of the lower bounds is 3.43 when the memory access time is 16 clock periods.

For the SP mode of execution, the timing diagram in Figure 5.1(b) applies when the memory access time is long. The time interval $t_2 - t_1$ is the turnaround time of a *Load* instruction. For small loops, the load distance $t_3 - t_1$ is shorter then the memory turnaround time. An arithmetic operation has to wait for the operands for a period $t_2 - t_3$. The execution time of each iteration includes this waiting period. One way to reduce the execution time is to increase the load distances with the techniques described in Section 5.2.3.

In the AE mode of execution, the long memory delay only increases the 'slippage' between the two processors' instruction streams. The long queues allows the access processor to run smoothly and load the operands for the execute processor. The differences among the execution times in the last row of Table 5.8 are mainly due to the instruction fetches which are delayed by the long memory access time. We believe that, for the AE mode of execution, most other loops would behave similarly. One obvious exception is the loops that involve read-after-write (RAW) hazards. As discussed in Section 5.2.4, the execution time of each iteration includes at least twice the memory access time. That section also shows two methods to avoid the degradation in performance caused by the hazards.

Remember that we assumed a conflict-free memory system in the above discussion. Table 5.4 shows that the memory module conflicts can degrade the PIPE performance in AE mode of execution. For example, the memory service rates of loops 8, 9 and 10 decreases drastically when we switch from the default parameter set to the empirical parameter set. We can improve the memory service rate at the cost of more circuit in the memory system. One method is to reduce the probability of memory module conflicts by organizing the memory into more modules. Another method is to service the memory requests out of order if necessary. Some sequencing mechanism is needed to insure that the requests are returned to the processors in their original requested order. The second method is currently being studied by Liou [Liou84]. Using the hand code of the LLL loops as benchmark, Liou observed that the second method can improve the PIPE performance by 40% to 50% when the memory is slow and the request rate is high

6. Conclusion

We have described the PIPE architecture and its implementation. Special features of PIPE include an instruction cache, architectural queues, a pipelined implementation, and decoupled execution. To evaluate the impact of these features, we have implemented a PIPE performance simulator, hand-coded a set of benchmarks, and simulated the execution of the benchmarks with various parameter sets and execution modes. The simulator proved very useful in our simulation work.

We have presented and analyzed simulation results based on the Lawrence Livermore Laboratories loops. Dependence of PIPE's performance on program characteristics were studied and optimizing techniques suitable for the PIPE architecture were discussed. In some loops, the small load distances and branch counts blocked instruction issue and limited PIPE performance. These problems were solved by code doubling or software pipelining. Read-after-write hazards were avoided by storing the responsible variables in registers. Balancing the work load in the two processors also contributed to the performance of decoupled execution. Table 5.5 showed an example in which the application of these simple optimizing techniques more than quadrupled PIPE performance.

Dependence of PIPE's performance on hardware speed and configuration was also investigated. The LDQ, OUTQ and BRQ were of size 4 in our parameter sets. It was observed that short queues sufficed for efficient execution of the benchmark programs we studied. When the memory was slow (longer memory access time) a longer REQQ and MLDQ were needed to buffer more memory transactions. These queues were of size 8 in our empirical parameter set. Figure 5.5 demonstrates the usefulness of architectural queues. It shows that, with enough queue sizes in the decoupled mode of execution, slower memory did not increase execution time. Our simulation results also demonstrated the advantages of decoupled architectures. Although the average speedup of AE mode over SP mode was only 1.473 for a fast memory, the value increased when the memory was slow, with a maximum speedup of 2.97 (Table 5.3). Further simulation indicated that the speedup could be as high as 3.43 when the memory access time was 16 clock periods (Tables 5.7 and 5.8). Simulation results also showed that the decoupled PIPE architecture performed very well on the benchmark programs.

Some interesting aspects of the PIPE architecture have not yet been studied. These include the instruction cache hit ratio, the queueing disciplines of the architectural queues, and the flow control mechanism of the busses. We need other, larger programs to study these aspects. Many program structures other than simple loops are used frequently in many applications and deserve careful study. Searching and sorting algorithms are two examples in which the branch conditions depend on the data being processed and may require synchronization between the access processor and the execute processor. Efficient implementation of procedure calls is also very important and worthy of study.

7. Acknowledgement

The authors would like to thank H. Young and P. B. Schechter who developed the Pascal compiler and the functional interpreter for the PIPE architecture, respectively. The authors are also indebted to K. Liou and J. Sayah for their valuable discussion during the course of this work.

References

[AlWo75] W. G. Alexander and D. B. Wortman, "Static and Dynamic Characteristics of XPL programs," *Computer*, November 1975

[BrWe83] W. C. Brantley and J. Weiss, "Organization and Architecture Trade-offs in FOM," *IEEE Workshop on Computer Systems Organization*, New Orleans, LA, pp. 139-143, March 1983.

[Brya83] R. M. Bryant, "SIMPAS 5.0 User Manual," Computer Sciences Department, Univ. of Wisconsin-Madison.

[CGKP83] G. L. Craig, J. R. Goodman, R. H. Katz, A. R. Pleszkun K. Ramachandran, J. Sayah, J. E. Smith, "PIPE: A High Performance VLSI Processor Implementation," Technical Report #513, Computer Sciences Department, University of Wisconsin-Madison, September 1983.

[CoSt81] E. U. Cohler, and J. E. Storer, "Functionally Parallel Architecture for Array Processors," *Computer*, Vol. 14, No. 9, pp. 28-36, September 1981.

[Cray79] "CRAY-1 Computer Systems, Hardware Reference Manual," Cray Research, Inc., Chippewa Falls, WI, 1979.

[Flyn66] M. J. Flynn, "Very High-Speed Computing Systems," *Proceedings of the IEEE*, Vol. 54, No. 12, pp. 1901-1909, December 1966.

[Good83] J. R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic" *Tenth Annual Symposium on Computer Architecture*, June 1983.

[Kogg81] P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill, 1981.

[Liou84] K. J. Liou, "Design of Pipelined Memory Systems for PIPE," Ph.D. dissertation, University of Wisconsin-Madison, Computer Sciences Department, in preparation, 1984.

[McMa72] F. H. McMahon, "FORTRAN CPU Performance Analysis," Lawrence Livermore Laboratories, 1972.

[MeCo80] C. Mead and L. Conway, *Introduction to VLSI Systems*, Addison-Wesley, 1980.

[PaDa76] J. H. Patel and E. S. Davidson, "Improving the Throughput of a Pipeline by Insertion of Delays," *Computer Architecture News (ACM-SIGARCH)*, Vol. 4, No. 4, pp. 159-164, January 1976.

[PaSe80] D. A. Patterson and C. H. Sequin, "Design Considerations for Single-Chip Computers of the Future," *IEEE Trans. on Computers*, Vol. C-29, No. 2, February 1980.

[PaSe81] D. A. Patterson and C. H. Sequin, "RISC I: A Reduced Instruction Set VLSI Computer," *Eighth Annual Symposium on Computer Architecture*, 1981.

[PlDa83] A. R. Pleszkun and E. S. Davidson, "A Structured Memory Access Architecture," *1983 International Conference on Parallel Processing*, Bellaire, MI, August 1983.

[Ples82] A. R. Pleszkun, "A Structured Memory Access Architecture," Computer Systems Group Report CSG-10, Coordinated Science Lab., Univ. of Illinois, Urbana, IL., October 1982.

[RaLi77] C. V. Ramamoorthy and H. F. Li, "Pipeline Architecture," *ACM Computing Surveys*, Vol. 9, No. 1, March 1977.

[Russ78] R. M. Russel, "The CRAY-1 Computer System," *Communications of the ACM*, Vol. 21, No. 1, pp. 63-72, January 1978.

[SPKG83] J. E. Smith, A. R. Pleszkun, R. H. Katz, and J. R. Goodman, "PIPE: A High Performance VLSI Architecture," *IEEE Workshop on Computer Systems Organization*, New Orleans, LA, pp. 131-138, March 1983.

[SmGo83] J. E. Smith and J. R. Goodman, "A Study of Instruction Cache Organizations and Replacement Policies," *Tenth Annual Symposium on Computer Architecture*, June 1983.

[Smit82] J. E. Smith, "Decoupled Access/Execute Computer Architectures," *Proc. of the Ninth Annual Symposium on Computer Architecture*, pp. 112-119, May 1982.

[Tane78] A. S. Tanenbaum, "Implication of Structured Programming for Machine Architecture," *Comm. ACM*, Vol. 21, March 1978.

[Youn84] H. Young, "Performance Evaluation of Decoupled Architectures by Compiler Construction," Preliminary proposal for doctoral dissertation, Computer Sciences Department, Univ. of Wisconsin-Madison, 1984.

Appendix A. Pascal source programs for the first 12 Lawrence Livermore loops

```
LLL1:  HYDRO EXCERPT
   1         q := 0;
   2         for k := 1 to 400 do
   3             x [k] := q + y [k] *
   4             (r * z [k+10] + t * z [k+11]);

LLL2:  MLR, INNER PRODUCT
   1         k := 1;
   2         while k <= 996 do
   3         begin
   4             tp[k] := z[k] * x[k] + z[k+1] * x[k+1] +
   5                 z[k+2] * x[k+2] + z[k+3] * x[k+3] +
   6                 z[k+4] * x[k+4];
   7             k := k + 5;
   8         end;

LLL3:  INNER PRODUCT
   1         q := 0;
   2         for k:= 1 to 1000 do
   3             q := q + z[k] * y[k];

LLL4:  BANDED LINEAR EQUATIONS
   1         L := 7;
   2         while (L <= 107) do
   3         begin       lw := L;
   4             j := 30;
   5             while (j <= 870) do
   6             begin   x[L-1] := x[L-1] - x[lw]*y[j];
   7                     lw := lw + 1;
   8                     j := j + 5;
   9             end;
  10             x[L-1] := y[5] * x[L-1];
  11             L := L + 50;
  12         end;

LLL5:  TRI-DIAGONAL ELIMINATION, BELOW DIAGONAL
   1         i := 2;
   2         while (i <= 1000) do
   3         begin   x[i] := z[i] * (y[i] - x[i-1]);
   4                 x[i+1] := z[i+1] * (y[i+1] - x[i]);
   5                 x[i+2] := z[i+2] * (y[i+2] - x[i+1]);
   6                 i := i + 3;
   7         end;
```

```
LLL6:  TRI-DIAGONAL ELIMINATION, ABOVE DIAGONAL
   1         j := 3;
   2         while j<=999 do
   3         begin
   4             i := 1000 - j + 3;
   5             x[i  ] := x[i  ] - z[i  ] * x[i+1];
   6             x[i-1] := x[i-1] - z[i-1] * x[i  ];
   7             x[i-2] := x[i-2] - z[i-2] * x[i-1];
   8             j := j + 3;
   9         end;

LLL7:  EQUATION OF STATE EXCERPT
   1         for m := 1 to 120 do
   2             x[m]:=u[m  ] + r*( z[m  ] + r*y[m  ] )
   3             + t*( u[m+3] + r*( u[m+2] + r*u[m+1] )
   4             + t*( u[m+6] + r*( u[m+5] + r*u[m+4] )));

LLL8:  P.D.E. INTEGRATION
   1    nl1 := 1;  nl2 := 2;
   2    for kx := 2 to 3 do
   3      for ky := 2 to 21 do
   4      begin
   5        du1[ky] := u1[kx,ky+1,nl1] - u1[kx,ky-1,nl1];
   6        du2[ky] := u2[kx,ky+1,nl1] - u2[kx,ky-1,nl1];
   7        du3[ky] := u3[kx,ky+1,nl1] - u3[kx,ky-1,nl1];
   8        u1[kx,ky,nl2] := u1[kx,ky,nl1]+a11*du1[ky]+
   9          a12*du2[ky]+a13*du3[ky]+sig*(u1[kx+1,ky,nl1]-
  10          2*u1[kx,ky,nl1]+u1[kx-1,ky,nl1]);
  11        u2[kx,ky,nl2] := u2[kx,ky,nl1]+a21*du1[ky]+
  12          a22*du2[ky]+a23*du3[ky]+ sig*(u2[kx+1,ky,nl1]-
  13          2*u2[kx,ky,nl1]+u2[kx-1,ky,nl1]);
  14        u3[kx,ky,nl2] := u3[kx,ky,nl1]+a31*du1[ky]+
  15          a32*du2[ky]+a33*du3[ky]+ sig*(u3[kx+1,ky,nl1]-
  16          2*u3[kx,ky,nl1]+u3[kx-1,ky,nl1]);
  17      end;

LLL9:  INTEGRATE PREDICTORS
   1      for i := 1 to 100 do
   2          px[1,i] := bm28*px[13,i] + bm27*px[12,i] +
   3                     bm26*px[11,i] + bm25*px[10,i] +
   4                     bm24*px[9,i] + bm23*px[8,i] +
   5                     bm22*px[7,i] + c0*(px[5,i]+px[6,i]) +
   6                     px[3,i];
```

```
LLL1Ø:  DIFFERENCE PREDICTORS
   1        for i := 1 to 1ØØ do
   2        begin
   3            ar      :=      cx[5,i] ;
   4            br      := ar - px[5,i] ;
   5            px[5,i] := ar           ;
   6            cr      := br - px[6,i] ;
   7            px[6,i] := br           ;
   8            ar      := cr - px[7,i] ;
   9            px[7,i] := cr           ;
  1Ø            br      := ar - px[8,i] ;
  11            px[8,i] := ar           ;
  12            cr      := br - px[9,i] ;
  13            px[9,i] := br           ;
  14            ar      := cr - px[1Ø,i];
  15            px[1Ø,i]:= cr           ;
  16            br      := ar - px[11,i];
  17            px[11,i]:= ar           ;
  18            cr      := br - px[12,i];
  19            px[12,i]:= br           ;
  2Ø            px[14,i]:= cr - px[13,i];
  21            px[13,i]:= cr           ;
  22       end;

LLL11:  FIRST SUM
   1        for k := 2 to 1ØØØ do
   2            x[k] := x[k-1] + y[k];

LLL12:  FIRST DIFF
   1        for k := 1 to 999 do
   2            x[k] := y[k+1] - y[k];
```